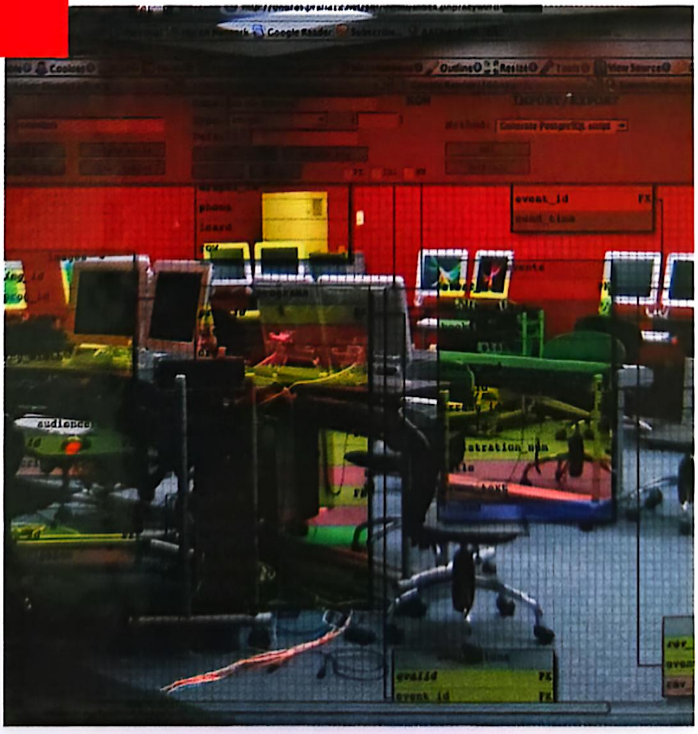
professional
practic

Marin Fotache

SQL

Dialecte DB2, Oracle, PostgreSQL și SQL Server

Ediția a II-a



POLIROM

Marin Fotache

SQL. Dialecte DB2, Oracle, PostgreSQL şi SQL Server

Marin Fotache (n. 1966) este absolvent al Universității „Al.I. Cuza" din Iași, Facultatea de Economie și Administrarea Afacerilor, specializarea Finanțe și Contabilitate. Este profesor la Catedra de Informatică Economică a aceleiași facultăți (www.feaa.uaic.ro). Dintre lucrările publicate: *SQL. Dialecte DB2, Oracle și Visual FoxPro* (Polirom, 2001), *Visual FoxPro. Ghidul dezvoltării aplicațiilor profesionale* (în colaborare, Polirom, 2002); *Oracle 9i2. Ghidul dezvoltării aplicațiilor profesionale* (în colaborare, Polirom, 2003) și *Proiectarea bazelor de date. Normalizare și postnormalizare. Implementări SQL și Oracle.* Este autorul a numeroase articole de specialitate publicate în revista *NetReport*.

© 2001, 2009 by Editura POLIROM

www.polirom.ro

Editura POLIROM
Iași, B-dul Carol I nr. 4; P.O. BOX 266, 700506
București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37;
P.O. BOX 1-728, 030174

Descrierea CIP a Bibliotecii Naționale a României:

FOTACHE, MARIN

SQL, dialecte DB2, Oracle, PostgreSQL și SQL Server / Marin Fotache. – Ed. a 2-a rev. și adăug. – Iași: Polirom, 2009

ISBN 978-973-46-1339-7

004.4

Printed in ROMANIA

Marin Fotache

SQL

Dialecte DB2, Oracle, PostgreSQL și SQL Server

Ediția a II-a revăzută și adăugită

POLIROM
2009

Cuprins

Prefață la ediția a II-a 11

Capitolul 1. Despre bazele de date 13

1.1. Nevoia de baze de date 13
1.2. Cum ne folosim de bazele de date 14
1.3. La început a fost fişierul (independent) 16
1.4. Ce este o bază de date? 18
1.5. Sisteme de gestiune a bazelor de date 20
1.6. Module, limbaje şi utilizatori ai SGBD-urilor 23
1.7. Utilizatorii bazelor de date 26
1.8. Modele de organizare a datelor în BD 28
1.9. Noțiuni preliminare în lucrul cu bazele de date 30

Capitolul 2. Modelul relațional de organizare a datelor 37

2.1. Structura 38
2.2. Restricții 42
2.3. Schema şi conținutul unei baze de date relaționale 51
2.4. SGBDR-uri şi servere de baze de date relaționale 63

Capitolul 3. Standarde şi dialecte SQL. Tipuri de date. Crearea tabelelor şi modificarea conținutului 73

3.1. Standardizarea SQL 75
3.2. Tipuri de date primitive SQL 82
3.3. Tabele şi restricții în SQL 87

3.4. Modificarea conținutului 114
3.5. Tranzacții 133

Capitolul 4. Limbaje de interogare. Algebra relațională 139

4.1. Caracterizare generală a limbajelor de interogare 139
4.2. Operatorii algebrei relaționale 141
4.3. Operatorii ansambliști 142
4.4. Operatorii relaționali 145
4.5. Notații și interpretări grafice ale interogărilor 168

Capitolul 5. Sintaxa de bază a comenzii SELECT 175

5.1. Trei clauze și un rezultat 175
5.2. Reuniuni, intersecții, diferențe, produse carteziene 183
5.3. Coloane-expresii 187
5.4. Ordonări 195
5.5. Operatorii BETWEEN, OVERLAPS și IN 198
5.6. Comparații inexacte. LIKE și SIMILAR 201
5.7. Joncțiuni interne 207
5.8. Sinonime locale și joncțiunea unei tabele cu ea însăși 215

Capitolul 6. Funcții SQL 219

6.1. Funcții pentru numere 219
6.2. Funcții pentru șiruri de caractere 225
6.3. Funcții pentru date calendaristice 231
6.4. Funcții pentru intervale 235
6.5. Conversii între tipuri de date 240
6.6. Alte funcții-sistem 249
6.7. Funcții-agregat 251

Capitolul 7. Grupări 269

7.1. Gruparea după un criteriu 269

7.2. Gruparea după două sau mai multe criterii ... 275
7.3. Gruparea după funcţii şi expresii ... 288
7.4. Clauza HAVING ... 297
7.5. Diviziunea relaţională ... 304

Capitolul 8. NULL-ităţi, joncţiuni externe şi structuri alternative ... 307

8.1. NULL-i în tabele ... 307
8.2. Operatorul IS NULL ... 313
8.3. Transformarea NULL-ilor ... 321
8.4. Joncţiunea externă ... 326
8.5. Structuri de control alternative ... 337

Capitolul 9. Subconsultări ... 357

9.1. Subconsultări în clauza WHERE. Operatorul IN ... 357
9.2. Subconsultări şi comparaţii în clauza WHERE ... 373
9.3. Subconsultări în clauza HAVING ... 380
9.4. Subconsultări în clauza FROM ... 394
9.5. Subconsultări scalare în clauza SELECT ... 428
9.6. Expresii-tabelă ... 433

Capitolul 10. Subconsultări corelate ... 443

10.1. Subconsultări corelate în clauza SELECT ... 443
10.2. Subconsultări corelate în clauza WHERE ... 462
10.3. Alte tipuri de corelări simple ... 475
10.4. Corelarea dublă ... 479

Capitolul 11. OLAP în SQL ... 485

11.1. Subtotaluri ... 485
11.2. Analize multidimensionale. Operatorii CUBE şi GROUPING SETS ... 500
11.3. Clasamente – soluţii clasice şi OLAP ... 515

11.4. Ferestre pentru funcțiile analitice 532
11.5. Comparații și ponderi 540

Capitolul 12. SQL, ierarhii și recursivitate 547

12.1. Autojoncțiuni și subconsultări 547
12.2. Interogări arborescente în Oracle 557
12.3. Interogări arborescente în MS SQL Server 567
12.4. Interogări arborescente în DB2 574
12.5. Interogări arborescente în PostgreSQL 577
12.6. Interogări mai puțin arborescente, dar tot recursive 580

Capitolul 13. Tabele, restricții și actualizări prin interogări SQL 591

13.1. Crearea tabelelor folosind interogări 591
13.2. Restricții și asertiuni pe bază de interogări 603

Capitolul 14. Tabele temporare și virtuale 633

14.1. Tabele temporare 633
14.2. Tabele virtuale în interogări 639
14.3. Probleme ale actualizărilor tabelelor-sursă pe baza modificării tabelelor virtuale 641
14.4. Restricții în tabele virtuale 651

Capitolul 15. Scheme, obiecte și utilizatori 653

15.1. Crearea de subscheme și acordarea de drepturi 653
15.2. Informații despre obiectele din (sub)scheme 664

Capitolul 16. Funcții, proceduri și pachete stocate 689

16.1. Funcții stocate 690
16.2. Cursoare 712
16.3. Funcții ce furnizează seturi de înregistrări 721

16.4. Funcții folosite ca valori implicite și restricții la nivel de atribut/înregistrare ... 730
16.5. Proceduri stocate ... 733
16.6. Pachete ... 748

Capitolul 17. Declanșatoare ... 753

17.1. Tipologia declanșatoarelor ... 753
17.2. Declanșatoare în PL/pgSQL ... 755
17.3. Declanșatoare în PL/SQL ... 766
17.4. Declanșatoare în Transact-SQL ... 778
17.5. Declanșatoare în SQL PL ... 787

Capitolul 18. SQL dinamic ... 793

18.1. SQL dinamic în PostgreSQL PL/pgSQL ... 793
18.2. SQL dinamic în PL/SQL ... 802
18.3. SQL dinamic în Transact-SQL ... 809
18.4. SQL dinamic în IBM DB2 SQL PL ... 816

Capitolul 19. Obiectual-relațional în SQL ... 825

19.1. Tipuri distincte și structurate de date ... 826
19.2. Colecții ... 838
19.3. Stocarea tipurilor-utilizator în tabele ... 846
19.4. Metode asociate tipurilor structurate în Oracle ... 854

Bibliografie ... 869

Prefață la ediția a II-a

Am recitit cu oarecare nostalgie prefața primei ediții de care mă despart opt ani și 20 de kilograme. Mi-am adus aminte și de simpatia pe care mi-o purta, la un moment dat, decanul unei facultăți surori (vitrege) celei în care lucram (și lucrez în prezent, deși „lucrat" poate nu e cel mai nimerit cuvânt), simpatie datorată tot unei prefețe, dar la a doua ediție a primei mele cărți, publicată la Editura Junimea [Fotache 1997]. Pasajul era plasat chiar la început și spunea că „ediția de față este una *revăzută*, deci câteva greșeli au fost corectate, dar și *adăugită*, prin urmare apar greșeli noi".

Cred că o asemenea introducere ar fi potrivită și acum. Cartea a căpătat proporții de la prima ediție, urmând un trend care îl surclasează pe cel urmat de greutatea autorului. Aproape nouă sute de pagini reprezintă o cifră frumoasă, dar și intimidantă, ducând cu gândul la faimoasa proporție 80/20: 80% dintre cititori vor folosi maximum 20% din ceea ce e scris în carte.

În afara noilor subiecte propuse, o diferență majoră o reprezintă și serverele de baze de date alese pentru exemplificarea ingredientelor SQL. Am renunțat, cu lacrimi în ochi, la Visual FoxPro, pe motiv de pensionare (a Fox-ului, eu mai am un pic!), și am inclus PostgreSQL - cel mai bun server BD de tip open source, în materie de SQL - și Microsoft SQL Server - pe motiv de popularitate. Câteva explicații suplimentare vă sunt oferite în capitolul 2.

Deși cartea este supraponderală (știți, probabil, că lucrările voluminoase sunt o obsesie a multor profesori trecuți de prima tinerețe), au rămas netratate câteva subiecte de interes pentru teoreticienii și practicienii bazelor de date, dintre care cel mai notoriu este XML (plus Xquery, LINQ etc.). Rămâne pe altădată.

Ce cărți dedicate SQL au mai apărut de la prima ediție? Destule pe plan internațional și nu prea multe la noi. Este adevărat, aproape orice carte dedicată bazelor de date tratează acest subiect în măcar un capitol, fie că este vorba despre autori români ([BDASEIG 2002], [Ionescu 2004], [Hurbean 2006]) sau străini ([Connoly & Begg 2002], [Garcia-Molina *et al.* 2002], [Date 2004/2005]). Dacă ne referim la volume tipărite în țara noastră, care să se ocupe exclusiv de SQL, pot să enumăr doar traduceri ale unor lucrări de inițiere ([Oppel 2006], [Fehily 2006]).

În explicarea noțiunilor și logicii comenzilor și opțiunilor SQL:1999, favoritele mele sunt cele două volume ale lui Jim Melton, primul fiind scris în colaborare cu Alan Simon ([Melton & Simon 2002], [Melton 2003]). La bibliografia primei ediții apăreau lucrările lui Paul Fortier ([Fortier 1999]) și Peter Gulutzan și Trudy Pelzer ([Gulutzan & Pelzer 1999]). Avantajul, dar, în egală măsură, și dezavantajul acestor lucrări este că se referă preponderent la versiuni ale standardului SQL, și mai puțin la implementările prevederilor din standard în sistemele de gestiune a bazelor de date existente pe piață. Neutralitatea lor este simultană cu lipsa de aplicabilitate

imediată. Pentru cei care vor să intre în intimitatea standardului, însă, lectura lor este necesară.

Dintre cărţile care raportează standardul la implementările „comerciale", v-aş recomanda-o, în primul rând, pe cea scrisă de Alex Kriegel şi Boris Trukhnov [Kriegel & Trukhnov 2008], dar şi [Groff & Weinberg 2002].

O categorie aparte o reprezintă cărţile în care sunt formulate soluţii SQL ingenioase la diferite probleme practice. La momentul primei ediţii, campionul era Joe Celko ([Celko 1997], [Celko 2000]). Ambele cărţi au acum noi ediţii ([Celko 2006] [Celko 2005]). Dintre „rivali" vă recomand cu dragă inimă şi excelentele volume [Molinaro 2006], [Faroult & Robson 2006] şi [Cumming & Russell 2007].

În final, îi mulţumesc lui Florin (Sîrbu) pentru răbdarea de a fi citit cartea şi pentru observaţii.

Vă salut şi sper să ne mai vedem până la viitoarea ediţie.

Marin Fotache
Iaşi, ianuarie 2009

Capitolul 1
Despre bazele de date

Se spune, pe drept cuvânt, că primele semne ale procesului de ramolire apar atunci când începi să te repeţi, confirmarea procesului fiind autocitarea, iar desăvârşirea autoplagierea (mai ales cea fără intenţie). Deşi tentative de a defini bazele de date apar în mai toate cărţile publicate, singur sau în colaborare, la Editurile Polirom ([Fotache 2001], [Fotache *et al.* 2002], [Fotache *et al.* 2003], [Fotache 2005]) şi Junimea [Fotache 1997], din raţiuni populiste voi apela pentru început, cu precădere, la secvenţe din două cursuri (*Instrumente software pentru afaceri* [Grama *et al.* 2006] şi *Baze de date I* [Fotache 2007]) care încearcă cu disperare să atragă atenţia studenţilor Facultăţii de Economie şi Administrarea Afacerilor, Universitatea „Al.I. Cuza", Iaşi, la disciplinele. Sunt vizate importanţa şi noţiunile fundamentale privind lucrul cu baze de date.

1.1. Nevoia de baze de date

Folosim bazele de date deoarece avem memoria prea scurtă şi stăm prost cu calculele. Ca şi în alte privinţe, avem nevoie de baze de date pentru că suntem limitaţi. Trăim într-o lume aşezată pe un morman de hârtii şi hârţoage şi ne este cu neputinţă să reconstituim ceea ce am făcut adineaori, darămite ieri, săptămâna trecută sau acum un an ori cinci. Necazul e că, de cele mai multe ori, trebuie să ştim nu numai ce-am făcut noi, dar şi câte ceva din ceea ce au făcut colegii şi partenerii de afaceri.

Simplificând şi exagerând nepermis lucrurile, am putea spune că există doi poli între care poate fi poziţionată orice problemă informatică. Pe de o parte, cel al chestiunilor interesante, nu neapărat cu formule şi calcule complexe, dar care reclamă un anumit grad de ingeniozitate în rezolvare - mult invocata şi aşteptata „fisă", iar pe de altă parte, cel care regrupează probleme în care complexitatea calculelor rareori depăşeşte nivelul celor patru operaţii aritmetice elementare - adunare, scădere şi încă două; în schimb, volumul informaţiilor şi zecile/sutele de moduri de regrupare şi agregare a lor sunt deconcertante.

Fără a face concurenţă vreunui manual de filosofie, putem spune că, din păcate, ca şi în viaţă, ponderea problemelor din a doua categorie – să le spunem plicticoase – este mult mai mare decât ponderea problemelor cu adevărat interesante. Aceasta ar fi vestea proastă. Vestea bună este că se câştigă enorm de mulţi bani din chestiunile plicticoase. O veste nici bună, nici rea ar fi că majoritatea problemelor pe care le are de rezolvat un informatician presupun elemente din ambele categorii.

De fapt, cei doi poli despre care vorbim sunt virtuali, fiind utili mai degrabă din raţiuni didactice şi pedagogice.

Cert este că, încă de la începuturile sale, informatica nu s-a confruntat numai cu efectuarea de calcule sofisticate, ştiinţifice, ci şi cu stocarea şi gestionarea unui volum de informaţii din ce în ce mai mare. Astfel, apariţia unor instrumente software dedicate gestiunii şi prelucrării datelor a fost doar o problemă de timp.

Prin urmare, avem nevoie de baze de date pentru a păstra, într-un format utilizabil, date şi informaţii legate de evenimente, tranzacţii etc. şi, la nevoie, de a le regăsi şi de a le prelucra după cum ne cer împrejurările. Bazele de date nu reprezintă singurul instrument de stocare şi prelucrare a informaţiilor. Şi într-un banal fişier .DOC (document Word, WordPerfect...), prezentare PowerPoint sau foaie de calcul (Excel, Lotus 1-2-3...) păstrăm date. Ca să nu mai vorbim despre pagini Web. Problema este că în documente, foi de calcul, prezentări, fişiere HTML etc. datele sunt slab structurate. Pentru a localiza informaţiile trebuie folosite instrumente de căutare care să depisteze prezenţa unor cuvinte-cheie, eventual în preajma unor alte cuvinte-cheie (cu o afacere de genul acesta s-au umplut de bani cei de la Google), iar rezultatele acestui gen de căutare/localizare sunt, de multe ori, iritante prin ambiguitatea şi volumul lor.

Încheiem acest miniparagraf cu altă pereche de veşti. Cea bună este că, la acest moment, bazele de date reprezintă cel mai bine structurat mod de păstrare şi scotocire a informaţiilor, motiv pentru care piaţa produselor de lucru cu bazele de date se exprimă în valori de ordinul miliardelor de dolari, nefiind prea multe semne că s-ar diminua. Vestea rea ţine de faptul că, dintre toate datele şi informaţiile pe care le vehiculăm/gestionăm/prelucrăm pe hârtie, la telefon, pe bandă sau disc magnetic, optic etc., doar o mică parte este preluată şi preluabilă în bazele de date.

1.2. Cum ne folosim de bazele de date

Mulţi dintre noi suntem datornici, unii chiar vânduţi, băncilor, aşa că putem rememora împreună câteva „imagini" din plata unei rate. După tradiţionala bineţe, lucrătorul de la ghişeul băncii ne cere un document de identitate şi apoi începe să privească atent pe ecran la o serie de meniuri şi ferestre, tastează numele sau codul numeric personal, iar apoi ne comunică suma de plată (rata), eventual câte rate mai avem sau alte informaţii legate de credit[1].

Meniurile, ferestrele şi rapoartele de pe ecran constituie *interfaţa* aplicaţiei informatice a băncii, aplicaţie la care este conectat lucrătorul de la ghişeu - vezi figura 1.1. Ceea ce nu se vede pe ecran şi este necunoscut lucrătorului (şi multora dintre noi) este că informaţiile afişate pe ecran sunt preluate dintr-un bazin de date aflat îndărătul uneia sau mai multor cutii de tablă, numite uneori calculatoare.

---

1. Este adevărat că sunt bănci la care informaţiile enumerate pot fi obţinute doar printr-un efort traumatizant, însă de dragul exemplului să ne imaginăm că aşa stau lucrurile.

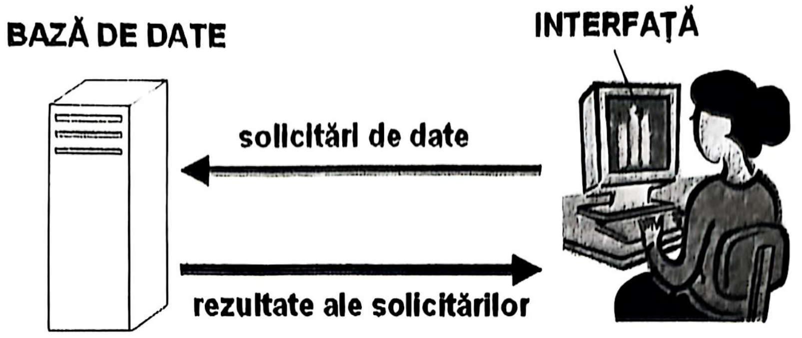


Figura 1.1. Schemă excesiv de simplistă a aplicațiilor ce utilizează baze de date

Figura 1.1 este cât se poate de simplistă. În realitate, baza de date se poate afla pe acelaşi calculator pe care se afişează meniurile, ferestrele şi rapoartele (situaţie frecventă în aplicaţii mici şi foarte mici realizate în Access sau FoxPro), dar, de cele mai multe ori, pe un alt calculator al cărui rol exclusiv este gestiunea datelor (serverul de baze de date). Pentru ca lucrurile să fie şi mai interesante, între calculatorul lucrătorului şi serverul pe care se află baza de date sunt interpuse alte *servere*, fiecare gestionând anumite module ale aplicaţiei - vezi figura 1.2.

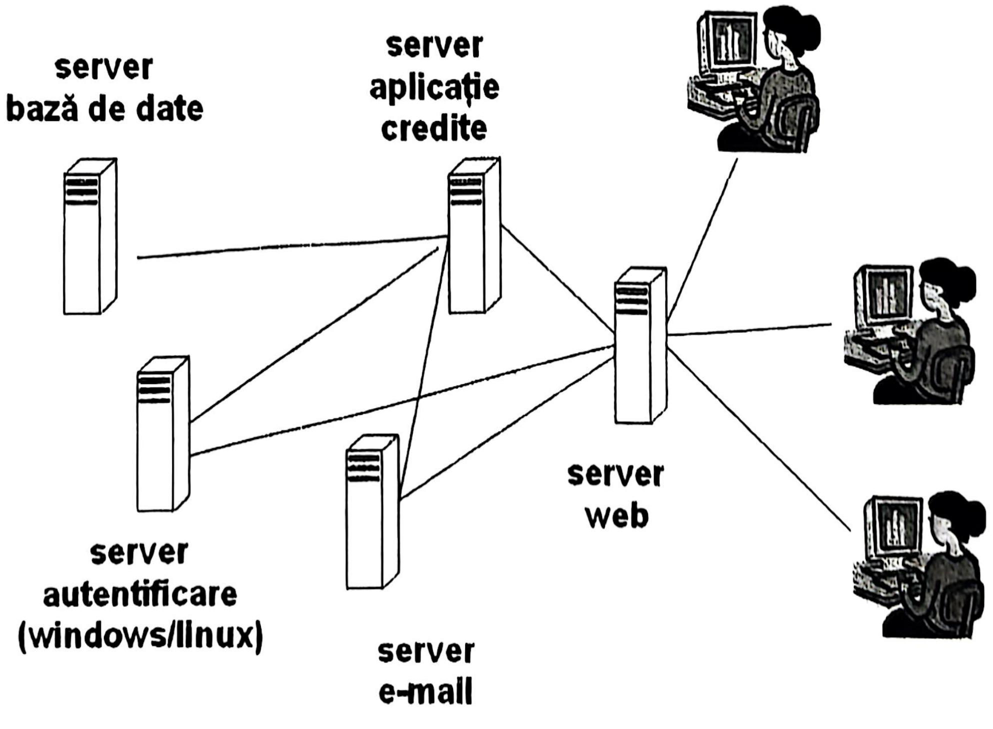


Figura 1.2. Schemă ceva mai complicată (dar tot inexactă) a aplicațiilor ce utilizează baze de date

Deşi ceva mai impresionantă, figura 1.2 conţine destule inexactităţi. De pildă, pot exista şi alte servere, în afara celor sugerate (de exemplu, servere pentru aplicaţii dedicate grupurilor de lucru (*groupware*), servere pentru gestiunea documentelor etc.). De fapt, aplicaţiile se descompun în procese şi servicii, fiecare server furnizând unul sau mai multe servicii. Apoi, trebuie spus că baza de date poate, la rândul său, să fie împrăştiată pe mai multe servere (vorbim despre baze de date centralizate, distribuite, replicate). În al treilea rând, nu toţi utilizatorii sunt

de sex feminin şi nu toate femeile din sistemul informatic al băncii au părul şaten prins în coadă/coc.

Ei bine, din toată figura, în această carte ne interesează numai serverul de bază de date. Mai mult (de fapt, mai puţin!), dintre zecile de subiecte dedicate serverelor de baze de date, noi ne vom ocupa doar de câteva: cum creăm şi folosim o bază de date, şi, mai ales, cum o stoarcem de informaţii?

Există totuşi persoane care folosesc baze de date fără a avea nevoie de formulare, meniuri şi alte elemente de interfaţă. Se poate crea şi gestiona o bază de date şi în acest mod, căruia putem să-i spunem spartan (sau stoic). Însă acest tip de utilizatori trebuie să fie aşi (sau măcar valeţi) în ale bazelor de date. Or, o caracteristică esenţială a bazelor de date este accesibilitatea. O bază de date este cu atât mai valoroasă cu cât pot avea acces la informaţiile sale cât mai multe categorii de utilizatori (fireşte, fiecare cu drepturile şi îndatoririle sale).

1.3. La început a fost fişierul (independent)

Schema din figura 1.2 este valabilă doar de câţiva ani încoace, însă mai toate aplicaţiile informatice pentru organizaţii (firme, spitale, universităţi, bănci etc.) realizate în ultimii 50 de ani prezintă cel puţin două „straturi": interfaţa şi datele. Înainte de folosirea bazelor de date, datele erau organizate în fişiere independente gestionate prin programe scrise în limbaje precum COBOL, FORTRAN, C, Basic, Pascal etc.

În a doua parte a anilor '50, Departamentul Apărării al SUA a format un grup de specialişti pentru elaborarea unui limbaj destinat aplicaţiilor administrative, în care dificultatea majoră ţinea de volumul imens de resurse materiale şi financiare ce trebuia „chivernisit" şi pentru care erau necesare rapoarte dintre cele mai diverse. Pornind de la precursorul FLOWMATIC, grupul cu pricina a redactat specificaţiile limbajului care a fost considerat câteva decenii regele informaticii economice - COBOL (Common Business Oriented Language).

Arhitectura aplicaţiilor de acest tip - specifică nu numai COBOL-ului, ci multor limbaje din a III-a generaţie, denumită *flat-files architecture* - tradusă în româneşte drept *fişiere independente* - este reprezentată în figura 1.3.

Specific acestui mod de lucru, referit ca *file-based* sau *flat files* (fişiere independente), este faptul că fiecare dată (Data1, Data2, ... Data*n*) este descrisă (nume, tip, lungime) autonom, în toate fişierele în care apare. Mai mult, descrierea fiecărui fişier de date (câmpurile care-l alcătuiesc, tipul şi lungimea fiecăruia, modul de organizare - secvenţial, indexat, relativ etc.) este obligatorie în toate programele care îl „citesc" sau îl modifică. Între FIŞIER1, FIŞIER2, ... FIŞIER*m* nu există nicio relaţie definită explicit.

Spre exemplu, Data2 este prezentă în două fişiere de date, FIŞIER1 şi FIŞIER2. Dacă, prin program, se modifică formatul sau valoarea acesteia în FIŞIER1, modificarea nu se face automat şi în FIŞIER2; prin urmare, o aceeaşi dată, Data2, va prezenta două valori diferite în cele două fişiere, iar necazurile bat la uşă:

informațiile furnizate de sistemul informatic sunt redundante şi prezintă un mare risc de pierdere a coerenței.

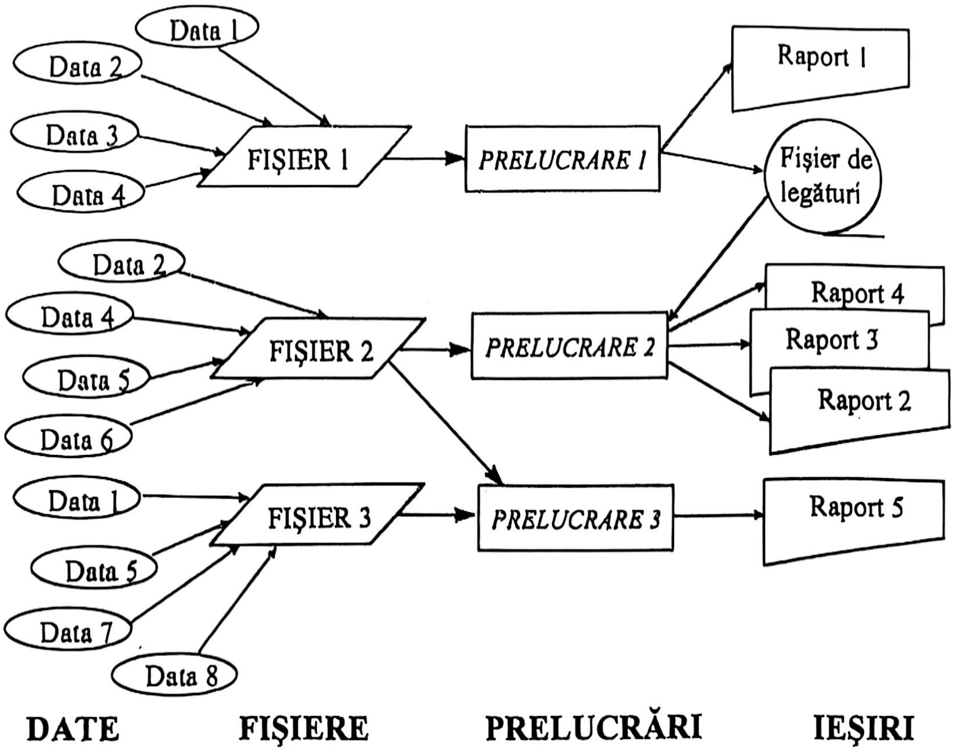


Figura 1.3. Sistem informatic bazat pe organizarea datelor în fişiere independente

Se poate proiecta un mecanism de menținere a integrității datelor, astfel încât actualizarea unei date într-un fişier să atragă automat actualizarea tuturor fişierelor de date în care aceasta apare, însă, în sistemele mari, care gestionează volume uriaşe de informații, implementarea unui asemenea mecanism este extrem de complexă şi costisitoare. În plus, fişierele de date sunt uneori proiectate şi implementate la distanțe mari în timp, în formate diferite: de exemplu, FIŞIER1 este posibil să fi fost creat cu ajutorul limbajului COBOL, FIŞIER2 în FORTRAN, iar FIŞIER3 în BASIC. În asemenea condiții, punerea în practică a mecanismului de menținere a integrității devine o utopie.

Chiar numai şi din cele prezentate mai sus, se pot desprinde câteva dezavantaje ale organizării datelor după modelul fişierelor independente:

- *redundanța şi inconsistența datelor*: o aceeaşi dată apare în mai multe fişiere; în aceste cazuri există riscul modificării acesteia într-un fişier fără a face modificările în toate celelalte fişiere;
- *dificultatea accesului*: într-o întreprindere, o aceeaşi informație este exploatată de mai mulți utilizatori. Spre exemplu, pentru departamentul care se ocupă cu gestiunea stocurilor, intrările de materiale trebuie ordonate pe magazii (depozite) şi repere, în timp ce pentru departamentul care se ocupă cu decontările cu partenerii de afaceri ai întreprinderii, intrările trebuie ordonate pe furnizori ai materialelor. Or, fişierele tradiționale nu facilitează accesarea

datelor după mai multe criterii, specifice diferiţilor utilizatori sau grupuri de utilizatori;
- *izolarea datelor*: când datele sunt stocate în formate diferite, este dificil de scris programe care să realizeze accesul într-o manieră globală a tuturor celor implicate în derularea unei tranzacţii;
- *complexitatea apăsătoare a actualizărilor*: o actualizare presupune adăugarea, modificarea sau ştergerea unor informaţii din fişiere. Cum prelucrările se desfăşoară în timp real, de la mai multe terminale (în mediile multiutilizator), pot apărea situaţii conflictuale atunci când doi utilizatori doresc modificarea simultană a unei aceleaşi date. Rezolvarea acestui gen de conflicte presupune existenţa unui program-supervizor al prelucrărilor, care este greu de realizat cu o multitudine de fişiere create la distanţă în timp şi în formate diferite;
- problemele de *securitate* ţin de dificultatea creării unui mecanism care să protejeze pe deplin datele din fişiere de accesul neautorizat;
- probleme legate de *integritatea datelor*: informaţiile stocate în fişiere sunt supuse la numeroase restricţii semantice. Toate aceste restricţii alcătuiesc mecanismul de integritate a datelor, deosebit de complex în mediile de lucru multiutilizator şi eterogene;
- inabilitatea de a obţine răspunsuri rapide la *probleme ad-hoc*, atât de frecvente în lumea afacerilor;
- *costul ridicat* ca urmare a gradului mare de redundanţă a datelor, eforturilor deosebite ce trebuie depuse pentru interconectarea diferitelor tipuri de fişiere de date şi pentru asigurarea funcţionării sistemului în condiţiile respectării unui nivel minim de integritate şi securitate a informaţiilor;
- *inflexibilitatea* faţă de schimbările ulterioare, ce sunt inerente oricărui sistem informaţional;
- modelarea indecvată a lumii reale.

Aceste dezavantaje sunt mai mult decât convingătoare, încât vă puteţi întreba dacă au existat inconştienţi care să-şi arunce banii pe apa... fişierelor independente. Ei bine, o serie de aplicaţii dezvoltate în anii '60 sau '70 au fost moştenite şi folosite până zilele noastre. De ce? Datorită sumelor consistente investite, care au putut fi amortizate (trecute pe costuri) doar în ani buni, chiar decenii. Un alt motiv a fost însă funcţionalitatea şi viteza unor asemenea aplicaţii, precum şi experienţa acumulată de o largă cateogorie de profesionişti în ale IT-ului.

1.4. Ce este o bază de date?

Bazele de date sunt, în general, percepute ca nişte uriaşe rezervoare informaţionale în care sunt turnate (sau, după caz, aruncate) tot soiul de cifre, şiruri de caractere, ba chiar imagini, texte etc., în speranţa că ar putea fi regăsite ulterior şi (re)ordonate, combinate şi grupate, în funcţie de nevoile utilizatorilor autorizaţi. În orice caz, o bază de date este *mare*, uneori imensă, *partajabilă* mai multor utilizatori

şi *persistentă,* adică poate fi stocată pe disc (sau orice suport) pe termen nelimitat ([Atzeni *et al.* 1999] pp. 3-4).

Sintagma *bază de date* apare pentru prima dată în titlul unei conferinţe organizate la Santa Monica (California) în 1964 de System Development Corporation. Consacrarea definitivă a termenului este marcată de publicarea în anul 1969, de către CODASYL, în cadrul unei conferinţe dedicate limbajelor de gestiune a datelor, a primului raport tehnic în care este prezentat conceptul de *bază de date*. Faţă de modelul fişierelor independente, noutatea o constituie existenţa unui *fişier de descriere globală a bazei,* astfel încât să se poată asigura independenţa programelor faţă de date, după cum o arată şi figura 1.4.

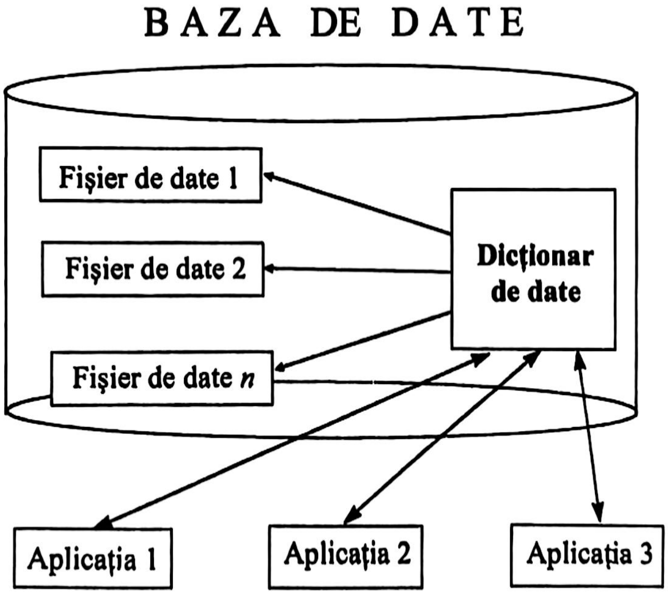


Figura 1.4. Schemă de principiu a unei baze de date

Avantajele organizării informaţiilor în baze de date decurg tocmai din existenţa acestui fişier de descriere globală a bazei, denumit, în general, *dicţionar de date* (alte titulaturi: *repertoar de date* sau *catalog de sistem*). Extragerea şi modificarea datelor sau altfel spus, lucrul cu fişierele de date, se derulează exclusiv prin intermediul dicţionarului în care se găsesc informaţii privitoare la structura datelor şi restricţiile îndeplinite de acestea.

O bază de date (BD) reprezintă un ansamblu structurat de fişiere, care grupează datele prelucrate în aplicaţiile informatice ale unei persoane, grup de persoane, întreprinderi, instituţii etc. Mai riguros, *BD poate fi definită ca o colecţie de date aflate în interdependenţă, împreună cu descrierea datelor şi a relaţiilor dintre ele,* sau ca *o colecţie de date utilizată într-o organizaţie, colecţie care este automatizată, partajată, definită riguros (formalizată) şi controlată la nivel central.*

Atunci când vorbim despre o bază de date, trebuie avute în vedere două aspecte fundamentale ale acesteia, *schema* şi *conţinutul*. Organizarea bazei de date se reflectă în schema sau structura sa, ce reprezintă un ansamblu de instrumente pentru descrierea datelor, a relaţiilor dintre acestea, a semanticii lor şi a restricţiilor la care sunt supuse. Ansamblul informaţiilor stocate în bază la un moment dat constituie conţinutul, instanţierea sau realizarea acesteia. În timp ce volumul

prezintă o evoluţie spectaculoasă în timp, schema unei baze rămâne relativ constantă pe tot parcursul utilizării acesteia. Corespunzător celor două aspecte complementare, schemă/conţinut, limbajele de programare dedicate bazelor de date se împart în limbaje de definire a datelor (DDL - Data Definition Language) şi limbaje de manipulare a datelor (DML - Data Manipulation Language). Limbajele uzuale dedicate bazelor de date, cum este SQL-ul, prezintă opţiuni atât pentru declararea structurii, cât şi pentru editarea conţinutului şi consultarea/interogarea bazei.

1.5. Sisteme de gestiune a bazelor de date

Aşa cum pentru fişierele de tip document este nevoie, în scopul editării, de procesoare de texte (Microsoft Word, WordPerfect etc.) sau pentru fişierele de tip foi de calcul (.XLS, WK1 etc.) de programe de calcul tabelar (Excel, Lotus 1-2-3, Quattro Pro), şi o bază de date solicită pentru creare, utilizare şi administrare un software specializat, şi anume un *sistem de gestiune a bazei de date* (SGBD).

Apărute în anii '60, sistemele de gestiune a bazelor de date reprezintă un ansamblu de programe ce permit utilizatorilor să interacţioneze cu o bază de date, în vederea creării, actualizării şi interogării acesteia. SGBD-ul este cel care asigură şi supervizează: introducerea de informaţii în baza de date; actualizarea şi extragerea datelor din bază; autorizarea şi controlul accesului la date; păstrarea independenţei dintre structura bazei şi programe.

Obiectivul esenţial al unui SGBD este furnizarea unui mediu eficient, adaptat utilizatorilor care doresc să consulte sau să actualizeze informaţiile conţinute în bază. Bazele de date sunt concepute pentru a prelucra un volum mare de informaţii. Gestiunea acestora impune nu numai o structurare riguroasă a datelor, ci şi o raţionalizare a procedurilor de acces şi prelucrare.

Prin urmare, principalele funcţii ale unui SGBD vizează:

- descrierea ansamblului de date la nivel fizic şi conceptual;
- crearea (iniţializarea) şi exploatarea (consultarea şi actualizarea) bazei de date;
- controlul integrităţii bazei;
- confidenţialitatea informaţiilor conţinute în bază;
- accesul simultan al mai multor utilizatori la informaţii;
- securitatea în funcţionare;
- furnizarea unui set de comenzi şi instrucţiuni, necesare atât utilizatorilor pentru consultarea directă a bazei, prin intermediul unui limbaj de manipulare, cât şi programatorilor, pentru redactarea programelor de lucru cu baza de date;
- revizia şi restructurarea bazei;
- monitorizarea performanţelor.

Există diferenţe considerabile între SGBD-urile aflate azi pe piaţă, în privinţa performanţelor şi preţului. Access sau Visual FoxPro, ambele produse Microsoft, sunt SGBD-uri mai modeste în privinţa dimensiunii bazei de date. Visual FoxPro nu are, nici măcar în versiunea 9, opţiuni de declarare a utilizatorilor şi grupurilor de utilizatori, cu atât mai mult opţiuni avansate de administrare. În schimb, este

foarte generos în realizarea de meniuri, formulare şi rapoarte (interfaţa aplicaţiilor), ceea ce l-a făcut mult timp preferatul informaticienilor în dezvoltarea de aplicaţii mici şi medii. Astăzi numele grele în domeniul bazelor de date sunt Oracle (Oracle), DB2 (IBM) şi SQL Server (Microsoft), iar cei mai serioşi competitori ai acestora sunt SGBD-urile *open source* cum ar fi PostgreSQL, MySQL etc. Vom reveni asupra acestui subiect.

Într-un sistem informatic ce utilizează BD, bazele de date sunt privite împreună cu SGBD-urile. Organizarea datelor poate fi analizată din mai multe puncte de vedere şi pe diferite paliere. De obicei, abordarea se face pe trei nivele: fizic sau intern, conceptual sau global şi extern - vezi figura 1.5.

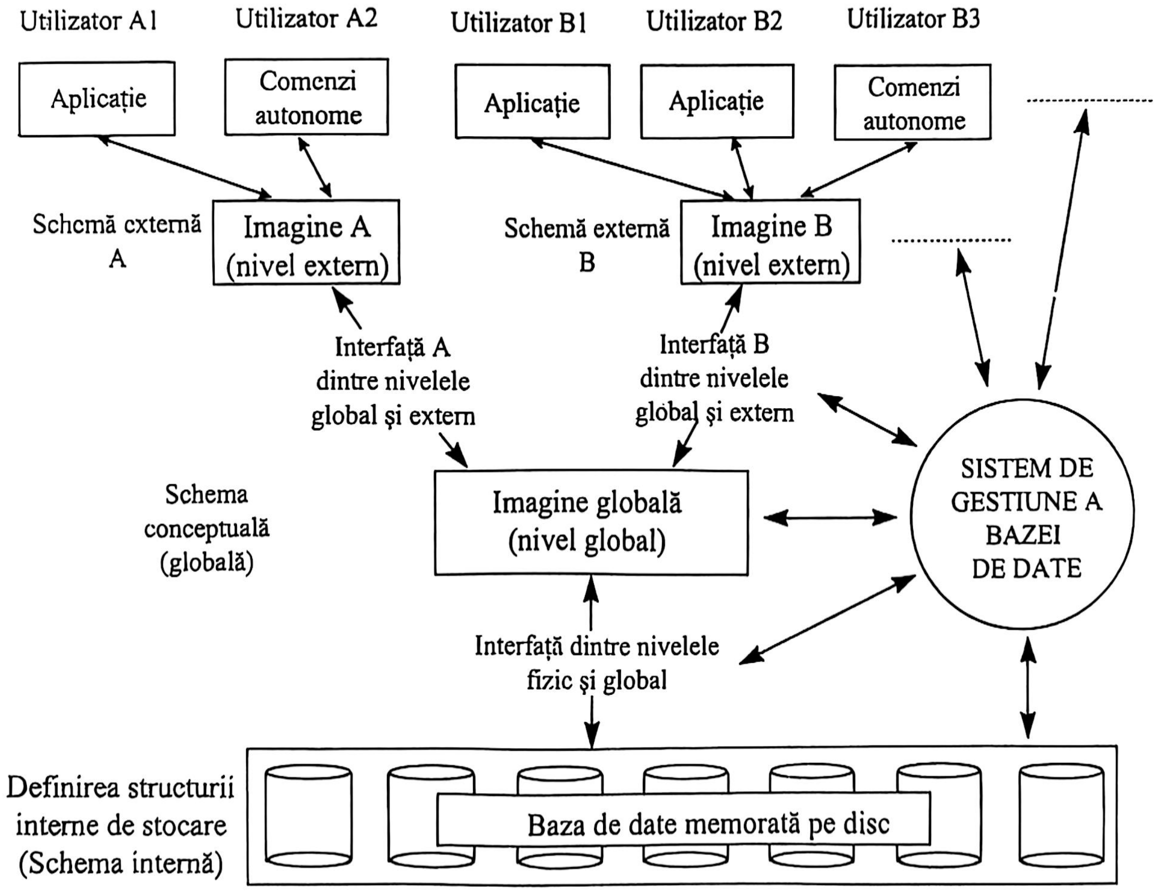


Figura 1.5. Schematizarea unui sistem de lucru cu o bază de date

Nivelul fizic (sau *intern*). Reprezintă modalitatea efectivă în care acestea sunt „scrise" pe suportul de stocare - disc magnetic, disc optic, bandă magnetică etc.

Nivelul conceptual (sau *global*). Este nivelul imediat superior celui fizic, datele fiind privite prin prisma semanticii lor; interesează conţinutul lor efectiv, precum şi relaţiile care le leagă de alte date. Reprezintă primul nivel de abstractizare a lumii reale observate. Obiectivul acestui nivel îl constituie modelarea realităţii considerate, asigurându-se independenţa bazei faţă de orice restricţie tehnologică sau echipament anume. Toţi utilizatorii îşi exprimă nevoile de date la nivel conceptual, prezentându-le administratorului bazei de date, acesta fiind cel care are o viziune globală necesară satisfacerii tuturor cerinţelor informaţionale.

Nivelul extern. Este ultimul nivel de abstractizare la care poate fi descrisă o bază de date. Structurile de la nivelul conceptual sunt relativ simple, însă volumul lor poate fi deconcertant. Iar dacă la nivel conceptual baza de date este abordată în ansamblul ei, în practică, un utilizator sau un grup de utilizatori lucrează numai cu o porţiune specifică a bazei, în funcţie de departamentul în care îşi desfăşoară activitatea şi de atribuţiile sale (lor). Simplificarea interacţiunii utilizatori-bază precum şi creşterea securităţii bazei sunt deziderate ale unui nivel superior de abstractizare, care este nivelul extern. Astfel, structura BD se prezintă sub diferite machete, referite, uneori şi ca subscheme, scheme externe sau imagini, în funcţie de nevoile fiecărui utilizator sau grup de utilizatori.

Posibilitatea modificării structurii la un nivel, fără a afecta structura nivelului sau nivelelor superioare, se numeşte *autonomie* a datelor stocate în bază, analizabilă pe două paliere.

Autonomia fizică reprezintă posibilitatea modificării arhitecturii bazei la nivel intern, fără ca aceasta să necesite schimbarea schemei conceptuale şi rescrierea programelor pentru exploatarea bazei de date. Asemenea modificări sunt necesare uneori pentru ameliorarea performanţelor de lucru (viteză de acces, mărimea fişierelor etc.). Tot autononomia fizică este cea care asigură portabilitatea bazei de date de pe un sistem de calcul pe altul fără modificarea schemei conceptuale şi a programelor.

Autonomia logică presupune posibilitatea modificării schemei conceptuale a bazei (din nevoia de rezolvare a unor noi cerinţe informaţionale) fără a rescrie programele de exploatare. Autonomia logică a datelor este mai greu de realizat decât autonomia fizică, deoarece programele de exploatare sunt dependente, în foarte mare măsură, de structura logică a datelor pe care le consultă şi actualizează, în ciuda existenţei dicţionarului de date. Fireşte, un element important îl reprezintă şi anvergura modificării schemei conceptuale.

Pe baza noilor elemente prezentate, putem să reluăm caracteristicile datelor stocate într-o BD[2]:

- partajabilitate - disponibilitate pentru un mare număr de utilizatori şi aplicaţii;
- persistenţă - existenţă permanentă, din momentul preluării în bază până în momentul actualizării sau ştergerii (ştergere cu sau fără arhivare);
- securitate - protejarea de accesul neautorizat, atât în ceea ce priveşte citirea şi copierea, cât şi modificarea şi ştergerea;
- validitate - referită şi ca integritate sau corectitudine - priveşte gradul de adecvare dintre datele din bază şi realitatea, procesele, tranzacţiile pe care le reflectă aceste date;
- consistenţă - de multe ori, procesele/tranzacţiile sunt preluate în bază sub forma mai multor entităţi (sau atribute). Aceste entităţi/atribute trebuie să fie în

2. Unii autori, precum [Atzeni *et al.* 1999], adaugă explicit drept caracteristică dimensiunea foarte mare a bazei de date. În cea mai mare parte a lucrărilor dedicate bazelor de date, această caracteristică este una implicită.

concordanţă unele cu celelalte, să respecte relaţiile existente între aspectele proceselor reale pe care se reflectă;
- nonredundanţă - pe cât posibil, o entitate din realitate ar trebui să aibă un singur corespondent în baza de date;
- independenţă - priveşte autonomia logică şi fizică evocată mai sus.

1.6. Module, limbaje şi utilizatori ai SGBD-urilor

Realizarea unui SGBD reprezintă un demers extrem de complex, ţinând seama de volumul datelor de stocat şi de prelucrat, numărul utilizatorilor conectaţi simultan la bază, viteza de onorare a solicitărilor informaţionale, precum şi cerinţele de integritate şi securitate. Majoritatea covârşitoare a celor ce lucrează cu bazele de date se descurcă onorabil fără a şti detaliile de organizare a SGBD-ului, ba chiar fără să fie conştienţi de existenţa acestuia. Alţii, precum administratorii de baze de date, sunt obligaţi să cunoască destul de multe despre organizarea fizică, indexuri, buffere, jurnal, drepturi ale utilizatorilor etc., iar alţii chiar lucrează în echipe care „fabrică" SGBD-uri, adică le proiectează, realizează anumite module, le testează etc.

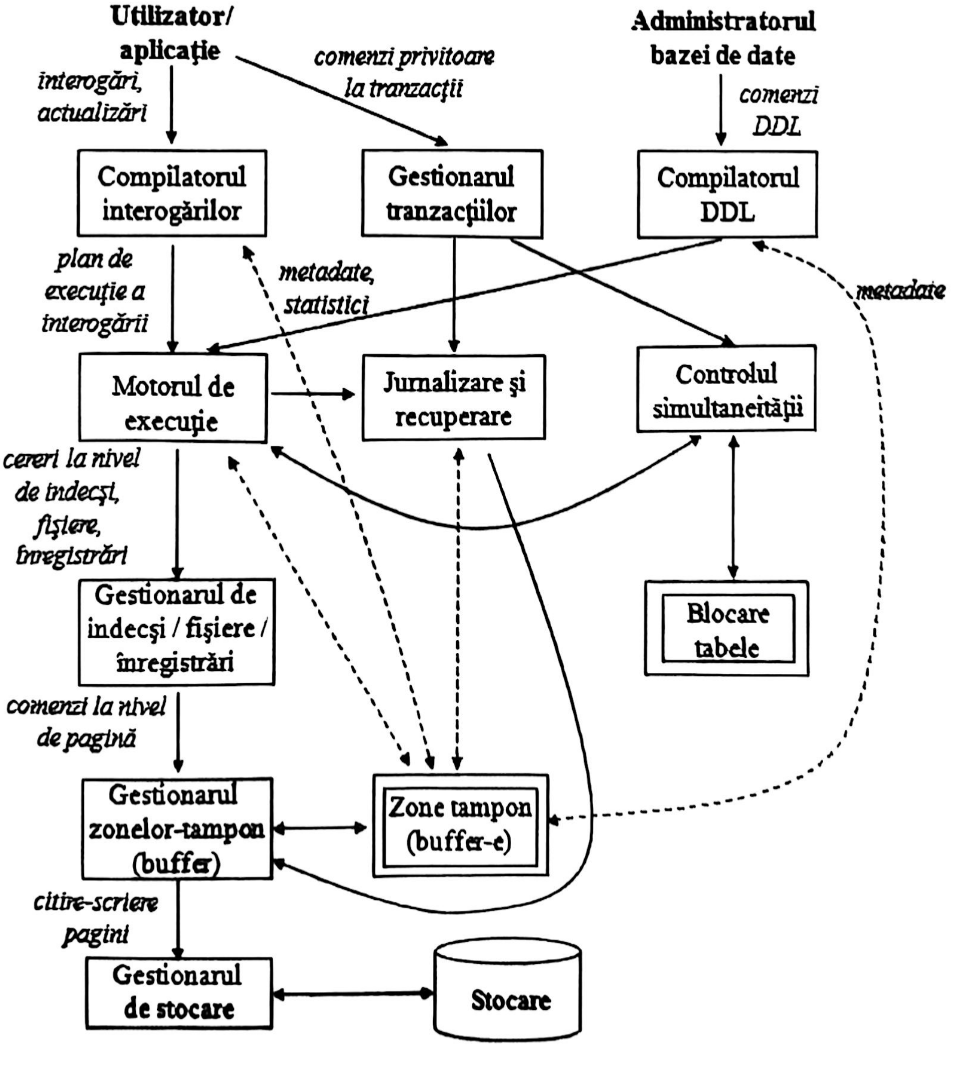


Figura 1.6. Structura unui sistem de lucru cu o bază de date (după [Garcia-Molina *et al.* 2002], p. 11)

Pentru a avea o idee generală despre modulele esenţiale ale unui SGBD apelăm la o schemă „clasică" prezentată în figura 1.6. După spusele autorilor - figuri foarte cunoscute în lumea bazelor de date -, dreptunghiurile simple reprezintă module ale sistemului, dreptunghiurile duble, structuri de date rezidente în memorie, săgeţile cu liniatură simplă indică fluxuri şi de date şi de control, în timp ce săgeţile punctate reprezintă fluxuri (numai) de date.

1.6.1. Module principale

Schema de mai sus este destul de impresionantă (cel puţin pentru mine), deşi este vorba despre o reprezentare simplificată a ceea se întâmplă în „inima" SGBD-ului. Sistemul nu „scrie" datele din memorie pe disc chiar imediat după preluarea lor, ci la anumite intervale de timp. De aceea, o componentă importantă, pe lângă gestionarul stocării, este gestionarul buffer-elor. Accesul rapid la date este imposibil fără indexuri, astfel încât în figură apare gestionarul indexurilor/fişierelor/înregistrărilor. Arbitrajul mai multor utilizatori/module ce lucrează simultan cu baza de date şi, implicit, rezolvarea eventualelor blocaje sunt asigurate de modulul de control al simultaneităţii (concurenţei).

Păstrarea integrităţii şi coerenţei bazei de date presupune gruparea operaţiunilor în tranzacţii pe principiul „totul sau nimic". Căderea uneia dintre operaţiunile tranzacţiei atrage în sine căderea întregii tranzacţii şi revenirea (restaurarea) bazei de date „pe starea" dinaintea tranzacţiei. Acest mecanism tranzacţional care presupune jurnalizarea cade sub incidenţa gestionarului de tranzacţii şi a modulului de jurnalizare şi recuperare.

1.6.2. Limbaje de definire a datelor

Figura 1.6 pune în evidenţă cele două compilatoare care corespund categoriilor clasice de limbaje pentru lucrul cu bazele de date - DDL (Data Definition Language), un limbaj destinat gestionării structurii bazei, şi DML (Data Manipulation Language), un limbaj pentru gestionarea conţinutului.

Arhitectura unei baze de date este specificată printr-o serie de definiţii redactate sub formă de instrucţiuni scrise în limbajul de definire a datelor - DDL. Execuţia acestor definiţii se materializează într-un ansamblu de tabele, tabele virtuale, proceduri stocate, secvenţe etc. care sunt memorate într-un fişier special, denumit *dicţionar* (sau repertoar) *de date*. Un dicţionar de date conţine aşadar *metadate*, adică date relative la alte date, fiind consultat înaintea oricărei citiri sau modificări a datelor din bază.

Principale funcţii ale DDL sunt:

- descrierea logică a bazei de date şi subschemelor proprii fiecărui grup de utilizatori;
- identificarea schemei, subschemelor şi diverselor agregări de date;
- specificarea fişierelor de date şi a legăturilor logice dintre acestea. Pe baza acestor specificaţii se poate realiza accesul la date chiar şi în condiţiile coexistenţei mai multor modele de organizare într-o aceeaşi BD;

- definirea restricțiilor semantice la care sunt supuse datele, ce se referă la ansamblul valorilor permise fiecărei date, eventual formula de calcul a unei date pe baza valorilor altor date. Respectarea acestor restricții asigură coerența bazei;
- definirea cheilor de acces rapid și a cheilor confidențiale (parolelor);
- definirea metodelor de „exploatare" a fișierelor ce vor fi utilizate în aplicații pentru selectarea înregistrărilor;
- definirea procedurilor speciale de criptare, în vederea generării cheilor de acces;
- definirea modalităților de indexare și localizare ale entităților;
- determinarea tipului unei date, *de bază* sau *derivată* (calculată printr-o expresie, pe baza valorilor altor date).

Dată fiind complexitatea structurilor de memorare și a metodelor de acces la acestea, la nivel elementar, fișierele de date și dicționarul de date sunt accesibile numai unui număr restrâns de *conaisseuri*. În schimb, pentru descrierea schemei BD, marea majoritate a limbajelor de interogare dispun de comenzi accesibile, ce pot fi utilizate și de neinițiați în informatică.

Cel mai important limbaj dedicat bazelor de date, SQL - despre care vom discuta începând cu al treilea capitol, prezintă comenzi DDL cum ar fi: CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE/ALTER/DROP VIEW, CREATE/DROP INDEX, CREATE/DROP PROCEDURE, CREATE/DROP TRIGGER, CREATE/DROP SEQUENCE etc.

1.6.3. Limbaje de manipulare a datelor

Prin manipularea datelor se înțelege efectuarea uneia dintre următoarele operațiuni:
- extragerea unor date din bază (consultare);
- scrierea unor date noi în bază (adăugare);
- ștergerea datelor perimate sau eronate (uneori chiar și a celor corecte);
- modificarea valorilor unor date.

Un limbaj de manipulare a datelor (DML) este utilizat pentru a prelucra datele în funcție de structura lor. La nivel fizic, prin DML se vizează identificarea și implementarea unor algoritmi performanți de acces la date, în timp ce la nivel extern, DML trebuie să faciliteze dialogul utilizatorului cu baza în vederea obținerii informațiilor dorite.

În mod curent, termenul *consultare* sau *interogare* desemnează acțiunea de căutare și identificare a datelor necesare, dintre cele aflate în bază. Ansamblul instrucțiunilor DML pentru căutarea și pentru identificarea datelor constituie limbajul de consultare. Revenind la SQL, cele mai cunoscute comenzi DML sunt: INSERT (pentru adăugarea de înregistrări într-o tabelă), DELETE (pentru ștergerea de linii dintr-o tabelă), UPDATE (pentru modificarea valorii unuia sau mai multor atribute, pe una sau mai multe înregistrări ale unei tabele) și SELECT (pentru extragerea de informații din una sau mai multe tabele).

1.6.4. Limbaje de control al datelor

Un element mai puţin vizibil în figura 1.6 este grupul de comenzi prin care se declară utilizatorii, grupurile de utilizatori (profilurile), precum şi drepturile fiecărui utilizator/profil la obiecte din schema bazei (tabele, tabele virtuale, proceduri etc.). Aceste comenzi sunt arondate unui al treilea tip de limbaj destinat bazelor de date - DCL (Data Control Language). Dintre acestea, în standardul şi dialectele SQL, cele mai frecvente sunt: CREATE USER, DROP USER, CREATE/DROP ROLE, GRANT şi REVOKE.

1.7. Utilizatorii bazelor de date

Figura 1.6 (stânga-sus) ilustrează accesul la baza de date a utilizatorilor/aplicaţiilor care pot lansa fie interogări şi actualizări (adresate compilatorului de interogări), fie comenzi privitoare la tranzacţii (COMMIT, ce declară încheierea cu succes a unei tranzacţii, şi ROLLBACK, ce abandonează o tranzacţie). Tot în partea de sus, dar în dreapta, este reprezentat personajul-cheie în exploatarea unei baze de date - administratorul bazei de date. Cei mai puţin vizibili în figura 1.6 sunt cei care participă la realizarea unui SGBD, figura concentrându-se asupra celor care folosesc bazele de date şi SGBD-ul din momentul „punerii în funcţiune" a aplicaţiei.

1.7.1. Utilizatori mai mult sau mai puţin curenţi

Cei mai numeroşi utilizatori ai unei aplicaţii ce foloseşte baze de date sunt şi cei mai „nevinovaţi" în ale bazelor de date. Doamnele (şi/sau domnişoarele) din figurile 1.1 şi 1.2 au la dispoziţie meniuri şi formulare pentru a introduce operaţiuni şi rapoarte pentru a extrage informaţiile necesare. Fluxurile dintre ecran/claviatură/mouse şi baza de date (locul unde se află, de fapt, informaţiile) nu sunt vizibile acestor utilizatori.

Realizarea aplicaţiilor care afişează pe ecran meniurile, formularele şi rapoartele şi care preiau datele introduse în baze cade în sarcina unei categorii profesionale destul de pestriţe denumită „dezvoltatori de aplicaţii cu baze de date". Zicem că este pestriţă, deoarece aici intră de-a valma: analişti şi proiectanţi de sisteme informaţionale, proiectanţi de baze de date (pot fi aceiaşi sau diferiţi de analiştii/proiectanţii de sisteme informaţionale), programatori de interfeţe .NET, Java, PHP etc., programatori de baze de date Oracle (PL/SQL), SQL Server (Transact-SQL), PostgreSQL (plpgSQL), DB2 (SQL PL) etc., testeri ş.a.m.d. Fireşte că, pentru ceea ce discutăm în cartea de faţă, programatorii de baze de date reprezintă o ţintă privilegiată.

Dezvoltatorii de aplicaţii sunt implicaţi în fazele de realizare a aplicaţiei (proiectare, programare, implementare/instalare). După darea în folosinţă, aplicaţiile pot suferi modificări de mai mică sau mai mare amploare (ca urmare a schimbărilor din legislaţia economică, regândirea modalităţilor de derulare a unor

operațiuni în cadrul organizației), modificări ce cad tot în (co)responsabilitea dezvoltatorilor. Deși sunt implicați în etapele de realizare a aplicației (unde pot formula cerințe, observații și proteste legate de modul în care funcționează modulele aplicației), utilizatorii curenți sunt destinatarii principali ai aplicației din momentul instalării (implementării) acesteia, cei care o vor folosi pentru a introduce operațiuni și a obține informații/rapoarte.

Între categoria „nevinovaților" și cea a dezvoltatorilor de aplicații cu baze de date există o varietate de utilizatori ce dispun de suficiente cunoștințe care să le permită obținerea de informații din bază sau chiar modificarea unei serii de date direct, fără mijlocirea aplicației. Prin urmare, există utilizatori care nu sunt nici administratori, nici dezvoltatori, dar care sunt în stare (și au drepturi suficiente) să lanseze interogări (fraze SELECT), comenzi de editare a înregistrărilor (INSERT/UPDATE/DELETE) sau să-și gestioneze „propriile" tabele, proceduri, tabele virtuale etc. Stilul acesta direct cu baza seamănă cu lucrul cu un ferăstrău electric fără apăratoare, deoarece chiar și în condițiile folosirii tranzacțiilor, câteva minute de neatenție pot antrena săptămâni întregi de migrene.

1.7.2. Administratorul bazei de date

Folosind limbajul mafiot, se poate spune că administratorul unei baze de date este „il capo di tutti capi" sau, în cel castrist, „el lider maximo". Administratorul unei baze de date este persoana responsabilă de sistem în ansamblul său. Rolul acestuia este determinant în:

- definirea arhitecturii bazei de date, realizată prin redactarea definițiilor care vor fi transformate de compilatorul DDL în tabele și alte obiecte stocate permanent în dicționarul de date;
- definirea modalităților în care va fi structurată memoria externă și a metodelor de acces la date;
- modificarea arhitecturii și organizării fizice a bazei de date;
- autorizarea accesului la date se acordă fiecărui utilizator sau grup (rol) al bazei de date, administratorul fiind cel care decide asupra datelor ce pot fi consultate și actualizate de fiecare utilizator sau grup de utilizatori;
- specificarea restricțiilor de integritate care sunt stocate pe disc și consultate de gestionarul bazei la fiecare actualizare.

În plus, administratorul bazei de date este cel care: asigură legătura cu utilizatorii; definește procedurile de verificare a drepturilor de acces și a procedurilor de validare a integrității datelor; definește strategia de salvare (înregistrarea copiilor de siguranță)/restaurare a bazei; monitorizează performanțele bazei și o adaptează la modificările ulterioare ale sistemului informațional.

Dat fiind că are mână liberă în crearea, utilizarea și optimizarea bazei, în acordarea de drepturi utilizatorilor, administratorul trebuie să fie nu numai un bun profesionist, ci și o persoană de caracter. Un supărăcios sau un frustrat poate nenoroci ireversibil tot capitalul informațional al firmei. Pe de altă parte, și

managerii ar trebui să se poarte cât mai frumos cu administratorul bazei de date (indiferent de ceea ce gândesc despre el).

1.8. Modele de organizare a datelor în BD

Nucleul unei baze de date îl reprezintă dicţionarul de date ce conţine structura bazei. Analiza, proiectarea şi implementarea structurii (schemei) bazei se realizează utilizând un model de date, ce reprezintă un ansamblu de instrumente conceptuale care permit descrierea datelor, a relaţiilor dintre ele, a semanticii lor, precum şi a restricţiilor la care sunt supuse.

Modelul datelor din BD este o reprezentare a obiectelor lumii reale şi a evenimentelor asociate lor. Presupune un demers de abstractizare care se concentrează asupra aspectelor esenţiale ale organizaţiei/aplicaţiei, furnizând conceptele şi notaţiile care vor permite utilizatorilor bazelor de date să comunice clar şi rapid informaţiile şi cunoştinţele lor despre datele organizaţiei.

O grupare „tradiţională" a modelelor utilizate în bazele de date delimitează trei categorii: modele logice bazate pe obiect, modele logice bazate pe înregistrare şi modele fizice. Din punctul nostru de vedere, sunt relevante numai nivelele conceptual şi extern de abstractizare a datelor; de aceea, vom prezenta, în linii mari, numai reprezentanţii principali ai primelor două categorii.

Modelul ierarhic. Primele software-uri (SGBD-uri) lucrau cu baze de date ierarhice. Structura datelor este prezentată sub forma unui arbore (vezi figura 1.7, partea superioară a arborelui fiind rădăcina; arborele este văzut „cu verdele în jos"). Un nod-tată poate avea mai multe noduri-fii. Un fiu nu poate exista independent de tatăl său. Legătura (reprezentată prin linie) se face exclusiv între tată şi fii. Orice fiu poate fi şi tată, deci poate avea, la rândul său, fii.

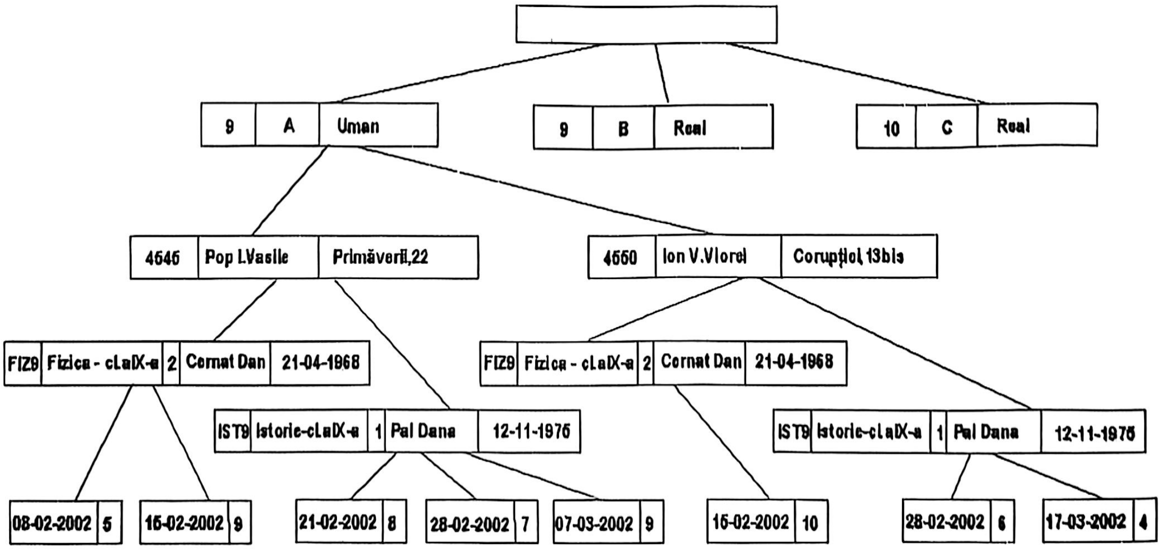


Figura 1.7. Arbore de structură specific modelului ierarhic

Astfel, în clasa a IX-a A (profil Uman) există doi elevi: Pop I. Vasile ce are numărul matricol 4545 şi domiciliul pe strada Primăverii nr. 22, iar al doilea, Ion V.

Viorel, are numărul matricol 4550 şi locuieşte pe strada Corupţiei nr. 13 bis. La fizică (profesor Cernat Dan, născut pe 21 aprilie 1968), Vasile a luat un 5 pe 8 februarie 2002 şi un 9 pe 15 februarie. La istorie (profesor Pal Dana, născută pe 12 noiembrie 1975) stă ceva mai bine, având un 8 (căpătat pe 21 februarie), un 7 (pe 28 februarie) şi un 8 (pe 7 martie). În acelaşi mod poate fi interpretată figura pentru Ion V. Viorel, cu menţiunea specială că istoria nu e punctul lui forte.

Modul în care înregistrările se înlănţuiesc presupune folosirea unor *pointeri* (un soi de adrese fizice), ce nu sunt reprezentaţi în figură (din considerente umanitare). Aflarea informaţiilor din baza de date presupune navigarea între înregistrări cu ajutorul pointerilor. În plus, modificarea structurii bazei de date presupune actualizarea înlănţuirii pointerilor, ceea ce este extrem de laborios. A altă problemă a acestui model o reprezintă imposibilitatea reprezentării cazului în care un copil este, pardon, „rezultatul" mai mulţi taţi. Situaţia nu este atât de scandaloasă precum vă închipuiţi, şi aceasta deoarece modelul ierarhic nu face nicio referire la mama fiului.

Figura 1.7 este relevantă pentru gradul de redundanţă impus de reprezentarea ierarhică. Codul, denumirea, numărul de ore ale fiecărei discipline, precum şi datele despre profesor, apar pentru fiecare elev, deşi acestea sunt comune la nivel de an de studii (numele disciplinei, numărul de ore) sau la nivel de clasă (profesorul titular).

Modelul reţea. Este o dezvoltare a modelului ierarhic, cu ajutorul căreia se pot reprezenta şi situaţiile în care un fiu „posedă" mai mulţi taţi. Înregistrările sunt privite în BD ca o colecţie de grafuri cu o structură asemănătoare celei din figura 1.8. Navigarea se face tot prin pointeri.

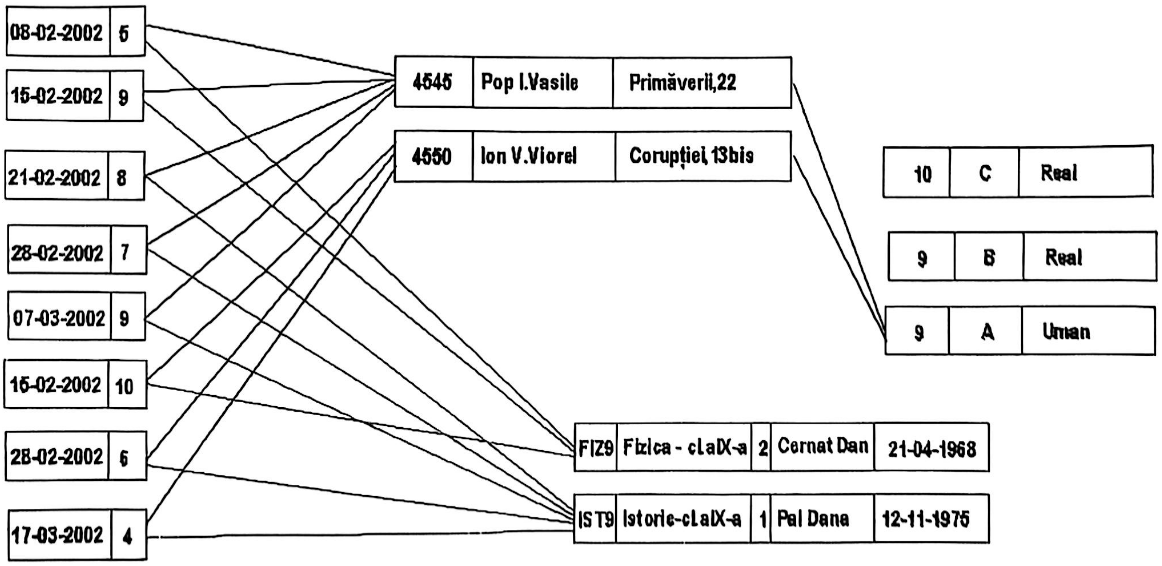


Figura 1.8. Graf specific reprezentării utilizând modelul reţea

Modelul reţea elimină o serie de probleme specifice modelului ierarhic. Spre exemplu, cazul ilustrat în figura 1.7 poate fi reprezentat după logica modelului reţea ca în figura 1.8. Se observă că fiecare disciplină (şi profesorul care o predă la clasa respectivă) apare o singură dată, fiind legată prin linii de toate notele obţinute

de fiecare dintre elevi. Legăturile dintre înregistrări se fac tot cu pointeri, iar modificarea înlănţuirii pointerilor rămâne o operaţiune foarte migăloasă.

Modelul relaţional a fost următorul în ordinea cronologică şi rămâne cel care domină de departe piaţa bazelor de date şi la acest moment, motiv pentru care îi rezervăm un capitol special, adică următorul.

Modelul obiectual. În programare, orientarea pe obiecte îşi are începuturile în anii '60, consacrându-se în anii '80 şi intrând în perioada de glorie după anii '90. Începând cu anii '90, orientarea pe obiecte capătă o importanţă majoră atât în analiză, cât şi în proiectare, reuşind să se constituie ca alternativă viabilă metodologiilor structurate. Pe baza acestui succes, s-a crezut că şi în materie de baze de date, modelul obiectual îl va surclasa pe cel relaţional până în 2000. Rezultatele sunt însă deprimante pentru suporterii OO-ului, piaţa SGBD OO fiind sub 6% din valoarea totală a pieţei bazelor de date[3].

Modelul relaţional-obiectual. Este un model mai recent ce încearcă să valorifice deopotrivă atuurile relaţionalului şi ale orientării pe obiecte. Deşi privit mai degrabă cu neîncrederea în cercurile teoreticienilor, acest model se impune încetul cu încetul datorită marilor producători de software dedicat bazelor de date.

Pe lângă aceste modele de organizare efectivă a datelor în baze, există şi modele pur teoretice, utile analiştilor şi proiectanţilor de sisteme informaţionale şi aplicaţii, dar care nu au fost încă implementate. Probabil cel mai cunoscut reprezentant al acestei categorii de *modele semantice* este modelul *entitate-relaţie*.

1.9. Noţiuni preliminare în lucrul cu bazele de date

Încercam să vă conving în paragraful 1.5 că nu putem lucra cu bazele de date decât prin intermediul unor sisteme de gestiune, asupra cărora ne vom opri şi în paragraful 2.4 (dedicat SGBD-urilor construite pe logica modelului relaţional). În continuare, dorim să lămurim câteva noţiuni legate de practica folosirii unui SGBD, şi anume: conexiune, sesiune, script, modul de program şi tranzacţie.

Conexiuni, sesiuni şi clienţi

Dacă vrem să adăugăm, să modificăm sau să ştergem informaţii dintr-o bază de date sau numai să „aruncăm un ochi" peste câteva informaţii aflate în bază, este necesar să stabilim o conexiune cu baza de date, prin intermediul SGBD-ului. Dacă vom reuşi, devenim *clienţii* bazei de date, calculatorul pe care lucrăm fiind *platforma-client*. SGBD-ul cu ajutorul căruia vom avea acces la bază este *serverul bazei de date*. Comanda folosită în multe SGBD-uri este chiar CONNECT. Pentru a fi luaţi în serios, trebuie să introducem la conectare un nume de utilizator şi o parolă pe care serverul să le recunoască; altfel, tocim claviatura degeaba - vezi figura 1.9.

3. Acest procent vehiculat la prima ediţie a cărţii a rămas valabil, nefiind exclus să fie, de fapt, supraevaluat.

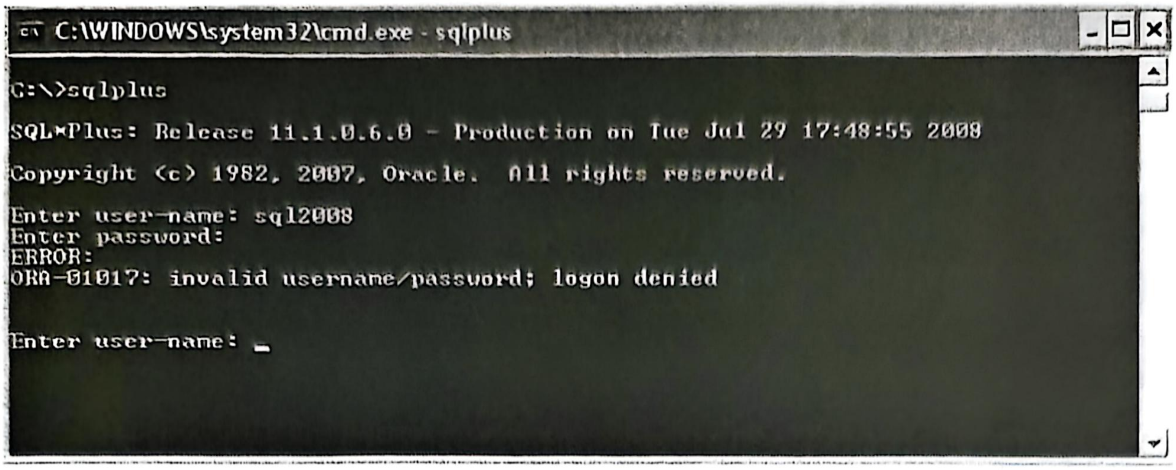


Figura 1.9. Tentativă eşuată de conectare la serverul Oracle 11g (din SQL*Plus)

Mă grăbesc să lămuresc câteva dintre eventualele *malentendu*-uri. Mai întâi, SGBD-urile actuale nu au o interfaţa atât de neguroasă. Doar administratorii/dezvoltatorii, din dorinţa de a impresiona, sau masochiştii (din alte raţiuni) mai lucrează în prezent în „modul text" (uneori, cele două categorii se intersectează în bună măsură). Restul oamenilor nu-prea-pricepuţi şi/sau normali preferă interfeţe mai prietenoase, după cum vom vedea nu peste mult timp. În al doilea rând, o conexiune la baza de date nu înseamnă nicidecum accesul la toată baza de date, ci doar la unele obiecte, pe care utilizatorul le poate „vedea" şi, eventual, modifica. Prin conexiune, avem acces la o fereastră a bazei. Conexiunea presupune ca, în prealabil, cineva să creeze utilizatorul, să-i atribuie o parolă şi să-i confere anumite drepturi pentru obiectele bazei de date. Atribuţiile acestui (semi)demiurg sunt îndeplinite de cele mai multe ori de duşmănoasa figură a administratorului bazei de date.

Între utilizatorul care se chinuie să se conecteze la bază şi baza propriu-zisă poate exista o distanţa de zeci, sute, mii... de calculatoare (uneori şi kilometri). Calculatorul pe care lucrează utilizatorul va adresa cererea de acces, iar SGBD-ul, ce dispune de un serviciu de ascultare (numit de obicei chiar *listener* - ascultător), preia cererea, verificând apoi corectitudinea numelui şi a parolei; dacă lucrurile sunt în regulă, se alocă un spaţiu special în memoria serverului pentru această conexiune în care utilizatorul urmează să se dezlănţuie. În plus, arhitecturile web actuale prezintă servere de aplicaţii care acţionează ca intermediari între utilizatori (clienţi) şi serverul bazei de date.

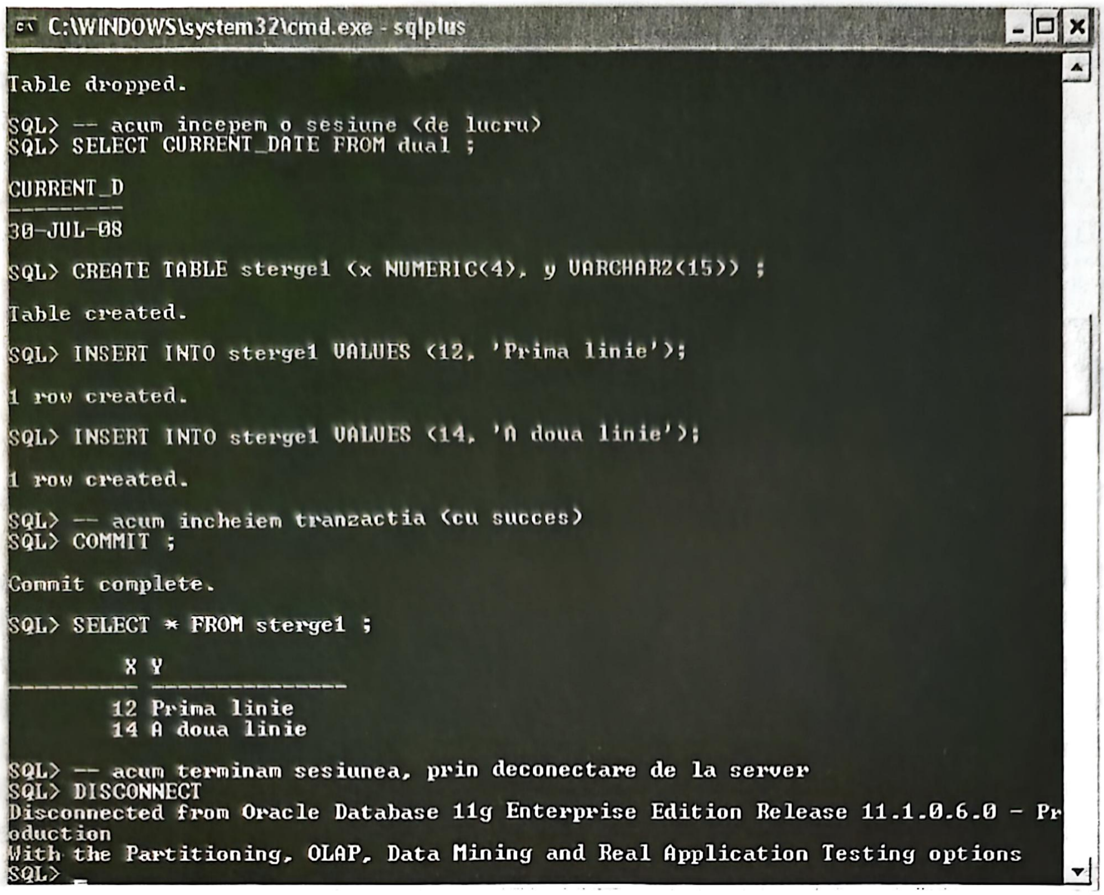


Figura 1.10. Exemplu simplist de conexiune/tranzacție/comenzi Oracle 11g (din SQL*Plus)

Deconectarea şi reconectarea la baza de date se realizează prin comenzi succesive DISCONNECT/CONNECT - vezi figura 1.10. O *sesiune* de lucru începe din momentul în care un client se conectează la un server de baze de date şi se termină în momentul deconectării de la server - vezi figura 1.10. Într-o sesiune pot fi lansate în execuție comenzi şi module de program.

După cum spuneam mai sus, ar fi destul de traumatizant să lucrăm cu o bază de date prin intermediul unei interfețe în mod text, precum cea din figurile 1.9 şi 1.10. De aceea, fiecare server de baze de date dispune de interfețe grafice atrăgătoare. Unele sunt parte integrantă a sofware-ului pentru baza de date. Este cazul MS SQL Server (*Management Studio*) - vezi figura 1.11.

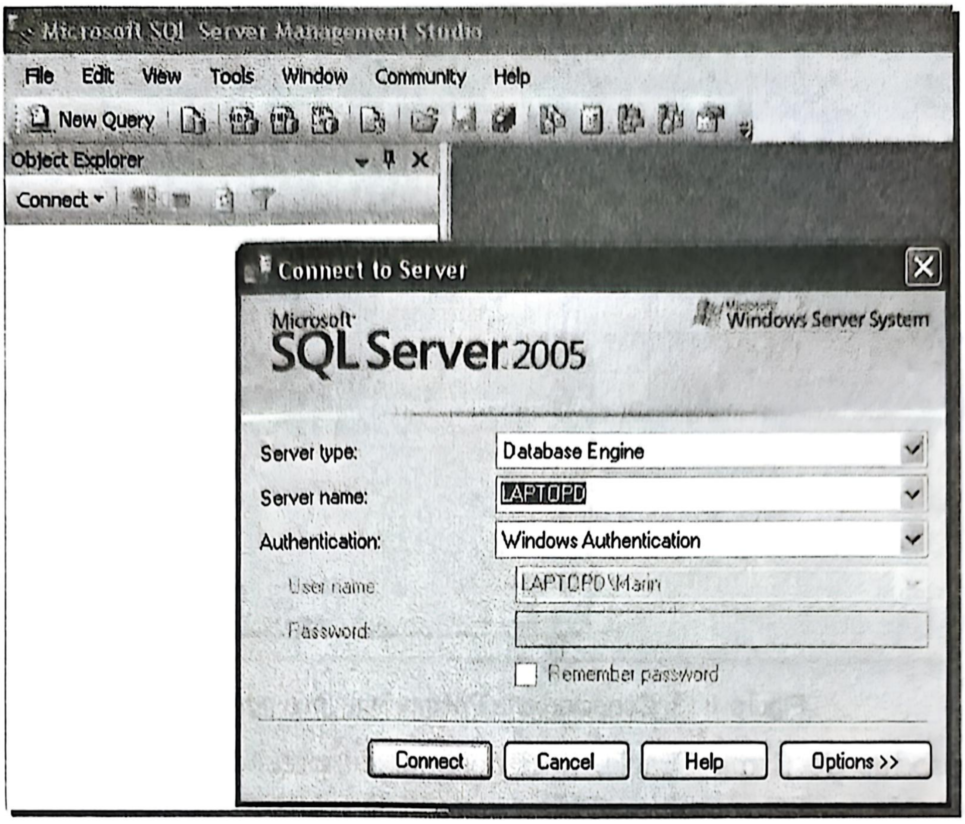


Figura 1.11. Conectarea la MS SQL Server (prin Management Studio)

Similar, IBM DB2 dispune de un modul numit *Control Center* prin care se poate realiza conectarea la baza de date - vezi figura 1.12. Pot fi instalate separat alte aplicații-client, cum ar fi *pgAdmin* dedicat PostgreSQL (vezi figura 1.13) sau *SQL Developer* pentru Oracle.

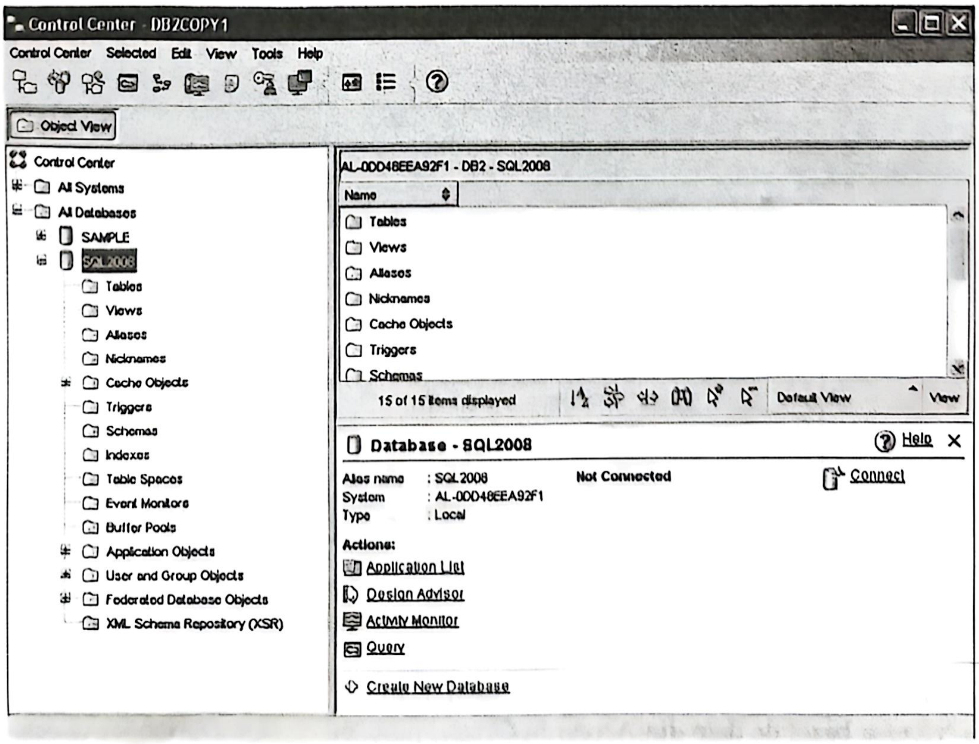


Figura 1.12. Conectarea la IBM DB2 (prin Control Center)

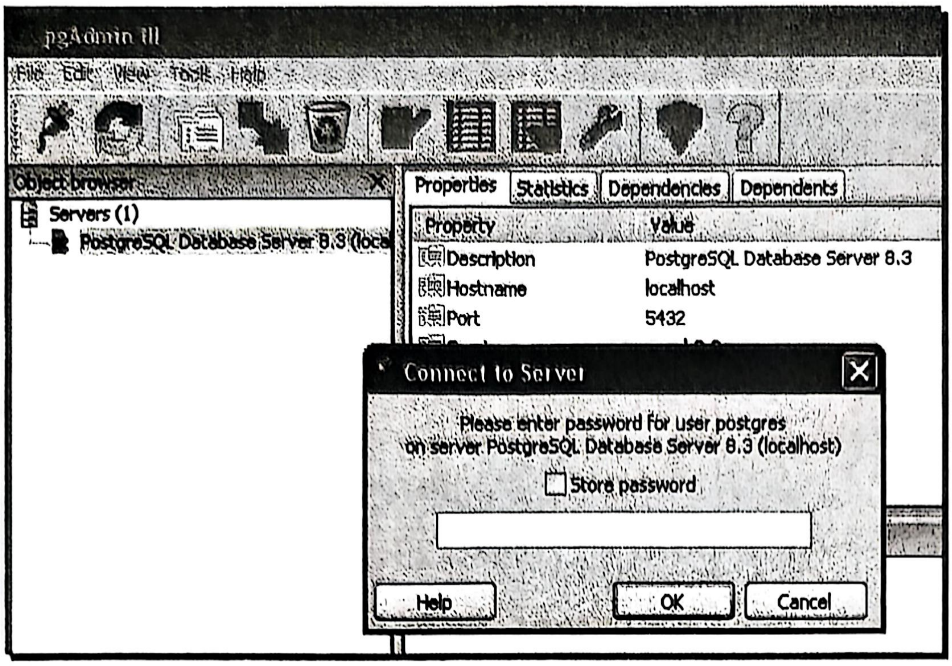


Figura 1.13. Conectarea la PostgreSQL (prin pgAdmin)

Deşi produs de firma Oracle, deci destinat cu precădere acestui server, SQL Developer este ceva mai ambiţios, fiind capabil să realizeze conexiuni şi cu baze de date create cu Microsoft Access - vezi figura 1.14[4]. Există şi clienţi „independenţi", unii dedicaţi unui singur SGBD (de exemplu, *SQL Navigator* produs de Quest Software, *SQLDetective* produs de Conquest Software, *PL/SQL Developer* produs de Allround Automations sunt destinaţi dezvoltării de aplicaţii cu baze de date Oracle), iar alţii asigură conexiunea la două sau mai multe SGBD-uri actuale, cum ar fi *TOAD* (Quest Software).

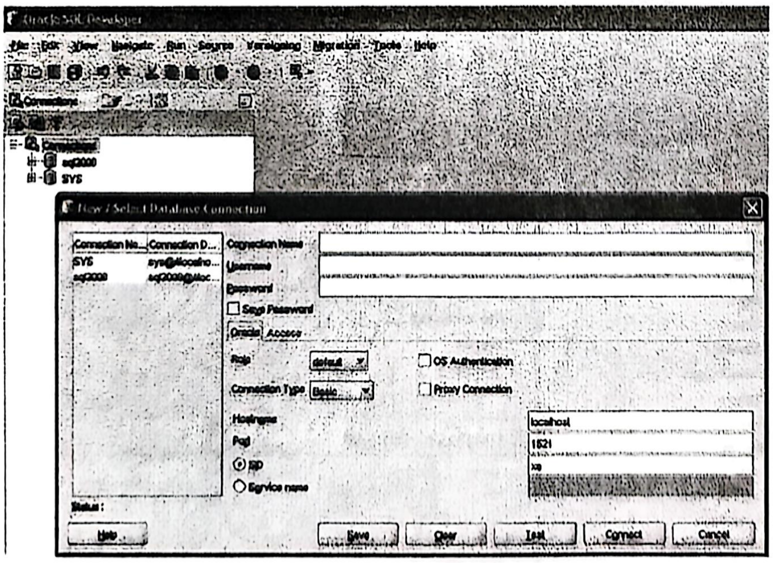


Figura 1.14. SQL Developer – un „client" pentru conectarea la baze de date Oracle sau Access

4. Nu cunosc niciun utilizator de Access care să folosească SQL Developer, însă atunci când se doreşte trecerea bazei de date din Access în Oracle, probabil că SQL Developer-ul este de un real folos.

Comenzi, scripturi şi module de program

Odată conectat la bază, un utilizator poate crea, modifica sau şterge obiecte - celebrele comenzi DDL (cum este CREATE TABLE în figura 1.10) -, sau consulta/modifica conţinutul bazei de date - comenzi DML (cum sunt comenzile INSERT şi SELECT din figura 1.10). Ar mai fi şi comenzile DCL prin care se creează utilizatori/grupuri şi se acordă/revocă drepturi la obiecte din bază, dar acestea sunt pentru administratori. Comenzile pot fi lansate direct, rând pe rând, sau pot fi grupate în module de program. În această carte vom folosi, de câteva ori, şi termenii *script* şi *program/modul*.

Pentru lucrarea de faţă, un script este un fişier (de obicei ASCII/text) care adună, una după alta, comenzi DDL şi/sau DML (sau chiar DCL). Un script *nu* conţine structuri de control alternative sau repetitive (IF-uri, LOOP-uri etc.), structuri ce constituie substanţa unui modul de program. Un modul de program este redactat într-un anumit limbaj (C, Java, Basic, PL/SQL, T-SQL etc.), în timp ce un script nu are nicio legătură cu un limbaj specific. Important este ca sintaxa comenzilor din script să fie acceptată de SGBD-ul pe care se lansează în execuţie.

Tranzacţii

Şi acesta este un subiect asupra căruia vom reveni pe parcursul volumului. Pentru început, să convenim că o tranzacţie este o succesiune de comenzi (de obicei, DML), care funcţionează pe principiul toţi pentru unul, unul pentru toţi sau, mai bine zis, toul sau nimic. De exemplu, grupăm într-o tranzacţie 50 de comenzi DML (inserare/modificare/ştergere), lansate individual, dintr-unul sau mai multe scripturi, şi/sau module de program. 49 merg strună, dar a 50-a generează o eroare în baza de date (încălcarea unei restricţii, să zicem). Din cauza acestei ultime comenzi buclucaşe, toate celelalte 49 de comenzi pot fi anulate, din raţiuni de solidaritate, printr-o comandă specială, ROLLBACK. Dacă şi a 50-a funcţionează cum trebuie, declarăm printr-o altă comandă - COMMIT - că tranzacţia s-a încheiat cu succes, iar modificările pot fi consemnate definitiv în bază. Lucrul cu tranzacţii este foarte important pentru păstrarea corectitudinii/integrităţii bazei de date, după cum vom vedea în câteva dintre capitolele viitoare.

Capitolul 2
Modelul relațional de organizare a datelor

Ca în atâtea alte domenii, și în teoria și practica bazelor de date se manifestă o dorință intensă de a scăpa de bătrâni, de a încerca și altceva. Modelul relațional este „bătrânul care nu vrea să se retragă". Apărut la începutul anilor '70, modelul s-a dovedit a avea oarecum soarta rockului acelorași ani, cu contestatari destul de vociferanți, dar cu înlocuitori mult mai palizi (de exemplu, discoul, punk), la propriu, la figurat și mai ales la trecerea timpului.

Afirmația de mai sus este incorectă politic, imposibil de demonstrat științific și degrabă aducătoare de ranchiună. Așa că nu are rost să vă enervați (mai bine vă puneți CD-ul cu albumul *London Calling* al celor de la The Clash). Lumea academică, în general, și lumea „liber-schimbistă" în ale bazelor de date fremăta, spre finalul anilor '80 și începutul anilor '90, la ideea că relaționalul și-a găsit nașul în modelul obiectual. Statisticile indicau că în anul 2000, odată cu sfârșitul lumii, bazele de date orientate pe obiecte vor depăși 50% din piață. După cum spuneam în primul capitol, nici în prezent procentul BDOO nu trece de 5 sau 6%.

N-aș vrea să înțelegeți că relaționalul nu va fi depășit într-o zi de obiectual sau de alt model (sau postmodel, cum este XML-ul). De asemenea, mărturisesc că nu am (deocamdată) contract cu modelul relațional. Afirmația la care subscriu este că modelul relațional s-a dovedit unul bun și, din multe puncte de vedere, este încă valabil pentru gestiunea datelor în firme și organizații, atât în forma sa originală, cât mai ales îmbogățit cu tipuri de date definite de utilizator (vezi capitolul 19).

Modelul relațional de organizare a datelor s-a conturat în două articole publicate în 1969 și 1970 de E.F. Codd, matematician la Centrul de Cercetări din San José (California) al firmei IBM[1]. În acel moment, tehnologia bazelor de date era centrată pe modelele ierarhic și rețea, care depind într-o mai mare măsură de organizarea internă a datelor pe suportul de stocare (celebrii pointeri pe care i-am pomenit fără a-i arăta în primul capitol). Codd a propus o structură de date tabelară, independentă de tipul de echipamente și software de sistem pe care este implementată, structură „înzestrată" cu o serie de operatori pentru extragerea datelor. Deși puternic matematizat, modelul relațional este relativ ușor de înțeles. Față de modelele ierarhice și rețea, modelul relațional prezintă câteva avantaje:

- propune structuri de date ușor de utilizat;
- ameliorează independența logică și fizică;

1. Extrase din lucrarea [Codd 1970] se găsesc la adresa http://www.acm.org/classics/nov95/. De asemenea, o excelentă analiză a articolelor lui Codd este propusă în [Date 1998].

- pune la dispoziţia utilizatorilor limbaje neprocedurale;
- optimizează accesul la date;
- îmbunătăţeşte integritatea şi confidenţialitatea datelor;
- permite tratarea unei largi varietăţi de aplicaţii;
- abordează metodologic definirea structurii bazei de date (normalizarea).

Deşi puternic contestat şi cu neajunsurile sale, modelul relaţional de organizare a bazelor de date rămâne cel mai utilizat. Cu foarte puţine excepţii, toate aplicaţiile software realizate pentru bănci, buticuri, universităţi (ordinea este, în ciuda aparenţelor, pur întâmplătoare) sunt realizate cu produse/instrumente ce gestionează baze de date relaţionale sau relaţional-obiectuale.

Un model de date are trei piloni: componenta *structurală*, adică modul în care, efectiv, la nivel logic, datele sunt stocate în bază, componenta de *integritate*, adică regulile/restricţiile ce pot fi declarate pentru datele din bază, şi o componentă *manipulatoare*, adică modul în care obţinem informaţii din bazele de date (ceea ce presupune o serie de operatori aplicabili uneia sau mai multor relaţii). Modelului relaţional îi este asociată teoria normalizării, care are ca scop prevenirea comportamentului aberant al relaţiilor în momentul actualizării lor, eliminarea datelor redundante şi înlesnirea înţelegerii legăturilor semantice dintre date[2]. Prezentul capitol va fi dedicat componentei structurale şi celei de integritate din modelul relaţional.

2.1. Structura

Modelul relaţional al datelor se poate defini printr-o serie de structuri de date (relaţii alcătuite din tupluri), operaţii aplicate asupra structurilor de date (selecţie, proiecţie, joncţiune etc.) şi reguli de integritate care să asigure consistenţa datelor (chei primare, restricţii referenţiale etc.)[3].

La modul simplist, o bază de date relaţională (BDR) poate fi definită ca un ansamblu de relaţii (tabele); fiecare tabelă (sau tabel), alcătuită din linii (tupluri), are un nume unic şi este stocată pe suport extern (de obicei, disc). La intersecţia unei linii cu o coloană se găseşte o valoare atomică (elementară, nedecompozabilă). O relaţie conţine informaţii omogene legate de anumite entităţi, procese, fenomene: CĂRŢI, STUDENŢI, LOCALITĂŢI, PERSONAL, FACTURI etc. Spre exemplu, în figura 2.1 este reprezentată tabela CLIENŢI.

---

2. Pentru detalii privind normalizarea, vezi (şi) [Fotache 2005], [Sitar-Tăut 2005].
3. Codd este cel care impune cele trei axe de discuţie ale unui model de date: structură, integritate şi manipulare. Vezi [Codd 1981].

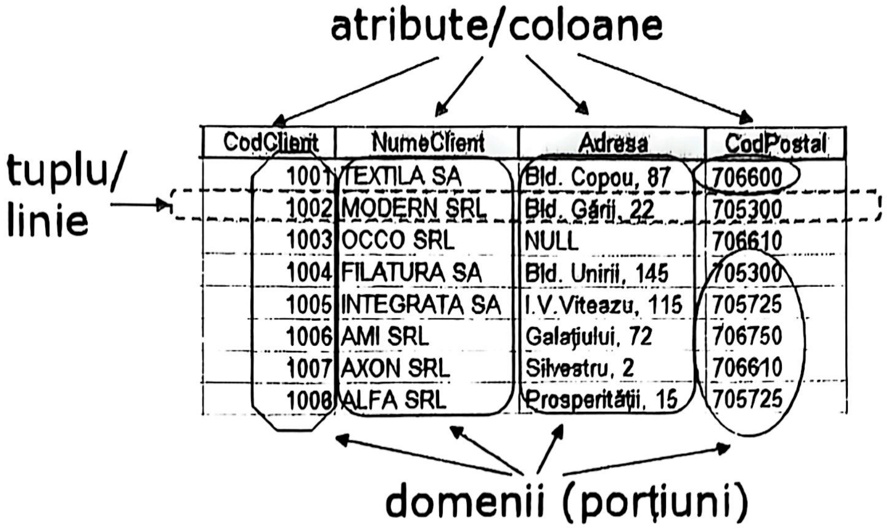


Figura 2.1. Relaţia (tabela) CLIENŢI

Teoria relaţională foloseşte termenul *relaţie*. Însă practica a consacrat termenul *tabelă* (engl. *table*). Un *tuplu* sau o *linie* este o succesiune de valori de diferite tipuri. În general, o linie (re)grupează informaţii referitoare la un obiect, eveniment etc., altfel spus, informaţii referitoare la o entitate: o carte (un titlu sau un exemplar din depozitul unei biblioteci, depinde de circumstanţe), un/o student(ă), o localitate (oraş sau comună), un angajat al firmei, o factură emisă etc. Figura 2.2 conţine al doilea tuplu din tabela CLIENŢI, cel referitor la firma MODERN SRL. Linia este alcătuită din patru valori ce desemnează: codul, numele, adresa şi codul poştal referitoare la clientul MODERN SRL.

1002	MODERN SRL	Bld. Gării, 22	705300

Figura 2.2. Un tuplu al tabelei CLIENŢI

Teoretic, orice tuplu reprezintă o *relaţie între clase de valori* (în cazul nostru, între patru clase de valori); de aici provine sintagma *baze de date relaţionale*, în sensul matematic al relaţiei, de asociere a două sau mai multe elemente. Fireşte, toate tuplurile relaţiei au acelaşi format (structură), ceea ce înseamnă că în tabela CLIENŢI, din care a fost extrasă linia din figura 2.2, fiecare linie este constituită dintr-o valoare a codului, o valoare a numelui, o valoare a adresei şi o valoare a codului poştal. Ordinea tuplurilor nu prezintă importanţă din punctul de vedere al conţinutului informaţional al tabelei.

Fiecare atribut este caracterizat printr-un *nume* şi un *domeniu* de valori pe care le poate lua. Domeniul poate fi definit ca ansamblul valorilor acceptate (autorizate) pentru un element component al relaţiei:

- într-o tabelă destinată datelor generale ale angajaţilor, pentru atributul Sex, domeniul este alcătuit din două valori: *Feminin* şi *Masculin;*
- pentru atributul Regiune, domeniul, deşi limitat, este ceva mai mare; valorile autorizate pot fi: *Dobrogea, Banat, Transilvania, Oltenia, Muntenia, Moldova.*

- domeniul atributului Jud este alcătuit din indicativele auto (două caractere) ale fiecărui judeţ (plus Bucureşti).
- domeniul atributului Judet este alcătuit din numele fiecărui judeţ (plus Bucureşti).
- domeniul unui atribut precum PretUnitar, care se referă la preţul la care a fost vândut un produs/serviciu, este cu mult mai larg, fiind alcătuit din orice valoarea cuprinsă între 1 şi 999999.99[4] lei (ceva mai noi).

Fireşte, în funcţie de specificul bazei de date, domeniul poate fi extins sau restrâns după cerinţe. De exemplu, la regiunile luate în discuţie mai sus, mai pot fi adăugate şi provincii precum: Bucovina, Maramureş, Crişana.

Şi acum, câteva senzaţii tari (urmează un pic de matematică)! Dacă notăm cu D_1 domeniul atributului CodClient, cu D_2 domeniul atributului NumeClient, cu D_3 domeniul pentru Adresa, şi cu D_4 domeniul atributul CodPostal, se poate spune că fiecare linie a tabelei CLIENŢI este un tuplu de patru elemente,

$$(v_1, v_2, v_3, v_4) \text{ unde } v_1 \in D_1, v_2 \in D_2, v_3 \in D_3, v_4 \in D_4$$

iar relaţia în ansamblu corespunde unui subansamblu din ansamblul tuturor tuplurilor posibile alcătuite din patru elemente, ansamblu care este produsul cartezian al celor patru domenii.

$$\bigotimes_{i=1}^{4} D_i$$

Generalizând, orice relaţie R poate fi definită ca un subansamblu al produsului cartezian de n domenii D_i:

$$R \subseteq D_1 \times D_2 \times D_3 \times \ldots \times D_n \text{ ,}$$

n fiind *gradul* sau *ordinul* relaţiei. Relaţiile de grad 1 sunt unare, cele de grad 2 - binare, ..., cele de grad n - n-are. Această definiţie pune în evidenţă aspectul constant al relaţiei, de independenţă în timp (se spune că în acest caz relaţia este definită ca un predicat).

O a doua definiţie abordează o relaţie R ca un ansamblu de m-uplete (m-tupluri) de valori:

$$R = \{t_1, t_2, \ldots, t_k, \ldots, t_m\}, \text{ unde } t_k = (d_{k1}, d_{k2}, \ldots, d_{ki}, \ldots, d_{kn})$$

în care:

d_{k1} este o valoare în D_1, d_{k2} este o valoare în D_2, ... , d_{kn} este o valoare în D_n;

n reprezintă *ordinul* lui R;

m - cardinalitatea lui R.

Pentru un (mic) plus de claritate, vezi figura 2.3. Reprezentarea sub formă de tabelă, deci ca ansamblu de tupluri, pune în evidenţă aspectul dinamic, variabil al relaţiei. Abordarea predicativă (prima definiţie) sau ansamblistă (a doua definiţie)

---

4. Punctul indică marca zecimală, cei doi de nouă care urmează punctului fiind asociaţi banilor (subdiviziunii leului).

reprezintă un criteriu important de delimitare a limbajelor de interogare a bazelor de date relaţionale.

R	D_1	D_2	D_3		D_i		D_n
t_1	d_{11}	d_{12}	d_{13}	...	d_{1i}	...	d_{1n}
t_2	d_{21}	d_{22}	d_{23}	..	d_{2i}		d_{2n}
t_3	d_{31}	d_{32}	d_{33}	...	d_{3i}	...	d_{3n}
	⋮	⋮	⋮		⋮		⋮
t_k	d_{k1}	d_{k2}	d_{k3}	...	d_{ki}	...	d_{kn}
	⋮	⋮	⋮		⋮		⋮
t_m	d_{m1}	d_{m2}	d_{m3}	...	d_{mi}	...	d_{mn}

Figura 2.3. Ilustrarea celei de-a doua definiţii a unei relaţii

Având în vedere cele două definiţii ale relaţiei, se poate reformula conceptul de *atribut*, ca fiind elementul care determină ansamblul de valori luate de fiecare domeniu al relaţiei ([Saleh 1994], p. 29). Precizarea fără ambiguitate a rolului jucat de fiecare domeniu în cadrul relaţiei impune ca numele atributelor să fie diferite. O relaţie poate fi simbolizată şi astfel:

$$R(A_1, A_2, ..., A_n) \text{ sau } (A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$$

unde A_i sunt atributele relaţie.

Reţinem corespondenţa noţiunilor *relaţie-tabelă*, *tuplu-linie* şi *atribut-coloană*.

Numărul tabelelor, atributele „adunate" în fiecare tabelă, domeniul fiecăruia dintre atribute prezintă diferenţe majore de la o bază la alta, chiar dacă uneori reflectă acelaşi tip de procese. Intrăm astfel în sfera proiectării bazelor de date, a dependenţelor şi normalizării.

Relaţia CLIENŢI conţine informaţii despre firmele cărora compania noastră le vinde produsele pe care le producem şi/sau comercializăm. Fiecare linie se referă la un singur client. În figura 2.1, pe a treia linie a tabelei, apare o valoarea curioasă notată NULL. Valoarea NULL este considerată o *metavaloare* şi indică faptul că, în acel loc, informaţia este *necunoscută*, *inexistentă* sau *inaplicabilă* (în acest caz nu cunoaştem adresa clientului MODERN SRL), fiind valabilă indiferent de tipul atributului (numeric, şir de caractere, date calendaristice). Valoarea NULL este diferită însă de valorile 0 sau spaţiu. Uneori, importanţa sa este (din păcate) majoră în expresii şi funcţii, după cum ştiu cei cu oarecare experienţă în limbajul SQL.

Principalele caracteristici ale unei relaţii pot fi sistematizate după cum urmează:

- În cadrul unei baze de date, o relaţie prezintă un nume distinct de al celorlalte relaţii.
- Valoarea unui atribut într-un tuplu oarecare este simplă (atomică sau elementară).

- Fiecare atribut are un nume distinct.
- Orice valoare a unui atribut face parte din domeniul pe care a fost definit acesta.
- Ordinea dispunerii atributelor în relaţie nu prezintă importanţă.
- Fiecare tuplu este distinct, adică nu pot exista două tupluri identice într-o aceeaşi relaţie.
- Ordinea tuplurilor nu influenţează conţinutul informaţional al relaţiei.

Structura unei baze de date mai cuprinde restricţii şi alte obiecte discutate în paragrafele 2.2 şi 2.3.

2.2. Restricţii

De ce ne interesează restricţiile într-o bază de date? Termenul *restricţie* (*regulă* sau *constrângere*) este oarecum iritant, atât pentru studenţi, cât şi pentru profesori, deoarece semnalează existenţa unor constrângeri instituite şi oarecum obligatorii, şi, de vreme ce sunt impuse, înseamnă că nu sunt prea plăcute (decât, în cel mai bun caz, pentru cel care le-a instituit). Partea cea mai enervantă este că respectarea restricţiilor este (supra)vegheată de o anumită autoritate înzestrată cu anumite instrumente de constrângere, de la bastoane de cauciuc, la creşterea şi scăderea impozitelor, salariilor, banilor de buzunar etc.

Ei bine, în bazele de date, regulile sunt ceva mai acceptabile. Cei care lucrează cu bazele de date sunt foarte interesaţi în declararea restricţiilor, pentru că, odată definite, de respectarea lor se va îngriji SGBD-ul. Esenţial este că, ajutaţi de reguli, putem creşte gradul de corectitudine şi de încredere pentru datele din bază. În continuare vor fi prezentate, pe scurt, cele mai importante restricţii definibile într-o BD relaţională: de domeniu, de atomicitate, de unicitate, referenţiale şi restricţiile-utilizator.

2.2.1. Restricţia de domeniu

După cum am văzut în paragraful anterior, un atribut este definit printr-un nume şi un domeniu. Orice valoare a atributului trebuie să se încadreze în domeniul definit. Există mai multe moduri de percepţie a acestei restricţii.

O parte dintre informaticieni substituie domeniul *tipului* atributului: numeric, şir de caractere, dată calendaristică, logic (boolean) etc. şi, eventual, lungimii (numărul maxim de poziţii/caractere pe care se poate „întinde" un atribut). După cum se observă, este luat în calcul numai aspectul sintactic al domeniului. Faptul că indicativul auto al unui judeţ (vezi plăcuţele de înmatriculare) poate fi una dintre valorile: IS, TM, B etc. reprezintă o restricţie de comportament sau, mai simplu, o restricţie definită de utilizator.

Cea de-a doua categorie priveşte domeniul deopotrivă *sintactic* şi *semantic*. Astfel, domeniul sintactic al atributului Jud (indicativul judeţului) este un şir de două caractere, obligatoriu litere (sau o literă şi un spaţiu, pentru Bucureşti), şi, chiar mai restrictiv, literele sunt obligatoriu majuscule. Din punct de vedere semantic, indicativul poate lua una dintre valorile: IS, TM etc.

Majoritatea SGBD-urilor permit definirea tuturor elementelor ce caracterizează domeniul (sintactic şi semantic) atributului Jud prin declararea tipului şi lungimii atributului şi prin aşa-numitele reguli de validare la nivel de câmp (*field validation rule*). Sunt însă şi produse la care domeniul poate fi definit explicit, sintactic şi semantic, dându-i-se un nume pe care vor fi definite atributele în momentul creării tabelelor.

2.2.2. Atomicitate

Conform teoriei bazelor de date relaţionale, orice atribut al unei tabele oarecare trebuie să fie atomic, în sensul imposibilităţii descompunerii sale în alte atribute. Implicit, toate domeniile unei baze de date sunt musai atomice (adică elementare). În aceste condiţii, se spune că baza de date se află în *prima formă normală* sau prima formă normalizată (1NF).

În prezent, atomicitatea valorii atributelor a devenit o ţintă predilectă a „atacurilor duşmănoase" la adresa modelului relaţional, pe motiv de imposibilitate a gestionării unor structuri de date mai complexe, specifice unor domenii ca proiectarea asistată de calculator, baze de date multimedia etc. Mulţi autori, dintre care merită amintiţi cu deosebire Chris J. Date şi Hugh Darwen, se opun ideii de atomicitate formulată de Codd.

La drept vorbind, singurul lucru cert despre atomicitate este relativitatea acesteia. Să reluăm un exemplu de atribut compus (nonatomic) - Adresa[5]. Fiind alcătuită din Stradă, Număr, Bloc, Scară, Etaj, Apartament, discuţia despre atomicitatea adresei pare de prisos, iar descompunerea sa imperativă. Trebuie însă să ne raportăm la obiectivele bazei. Fără îndoială că pentru BD a unei filiale CONEL, ROMTELECOM sau pentru poliţie, preluarea separată a fiecărui element constituent al adresei este foarte importantă. Pentru un importator direct însă, pentru un mare angrosist sau pentru o firmă de producţie lucrurile stau într-o cu totul altă lumină. Partenerii de afaceri ai acestora sunt persoane juridice, iar adresa interesează numai la nivel general, caz în care atributul Adresa nu este considerat a fi non-atomic.

Alte exemple de atribute ce pot fi considerate, în funcţie de circumstanţe, simple sau compuse: DataOperaţiuniiBancare (Data + Ora), BuletinIdentitate (Serie + Număr), NrÎnmatriculareAuto (privit global sau pe cele trei componente: număr, judeţ, combinaţie trei de litere).

O relaţie (tabelă) în 1NF nu trebuie să conţină atribute care se repetă ca grupuri (grupuri repetitive). Într-o altă formulare, toate liniile unei tabele trebuie să conţină acelaşi număr de atribute. Fiecare celulă a tabelei (intersecţia unei coloane cu o linie), altfel spus, valoarea unui atribut pe o linie (înregistrare), trebuie să fie atomică. Detalii despre atomicitate şi grupuri repetitive găsiţi (şi) în [Fotache 2005] şi [Sitar-Tăut 2006].

5. Este un exemplu dintre cele mai frecvente în majoritatea cărţilor despre baze de date.

2.2.3. Nenulitate

Modelul relaţional prevede ca, atunci când nu se cunoaşte valoarea unui atribut pentru o anumite entitate sau când pentru acel obiect, entitate, persoană etc. atributul este inaplicabil, să se folosească (meta)valoarea NULL. După cum am discutat, celui de-al treilea client din figura 2.1 nu i se cunoaşte adresa. Dacă am avea o tabelă PRODUSE cu atributele CodProdus, DenumireProdus, UM, Culoare, este posibil ca, pentru anumite sortimente, cum ar tricouri, cămăşi, jachete, atributul să fie important, în timp ce pentru altele, precum vodcă, cafea, lapte atributul Culoare să nu furnizeze nicio informaţie, adică să nu fie aplicabil. Iar dacă laptele poate fi doar alb, vodca chiar că nu are culoare (sunt foarte mulţi specialişti în acest domeniu, îi puteţi „interoga"!).

Într-o bază de date relaţională avem posibilitatea de a le impune unora dintre atribute (sau tuturor) să aibă întotdeauna valori specificate, altfel spus, le interzicem valorile nule, în timp ce altor atribute le permitem. Ca regulă, atributele importante, ce ţin de identificarea sau caracterizarea unei entităţi, proces, fenomen, precum şi cele implicate în calculul unor informaţii importante sunt declarate NOT NULL, atributele fără importanţă deosebită fiind mai „relaxate".

2.2.4. Unicitate

O relaţie nu poate conţine două linii identice (două linii care prezintă aceleaşi valori pentru toate atributele). Mai mult, majoritatea relaţiilor prezintă un atribut sau o combinaţie de atribute, care diferenţiază cu siguranţă un tuplu de toate celelalte tupluri ale relaţiei. *Cheia primară* a unei relaţii (tabele) este un atribut sau un grup de atribute care identifică fără ambiguitate fiecare tuplu (linie) al relaţiei (tabelei). După Codd, există trei restricţii pe care cheia primară trebuie să le verifice:

- *unicitate*: o cheie identifică un singur tuplu (linie) al relaţiei;
- *compoziţie minimală*: atunci când cheia primară este compusă, niciun atribut din cheie nu poate fi eliminat fără distrugerea unicităţii tuplului în cadrul relaţiei; în cazuri-limită, o cheie poate fi alcătuită din toate atributele relaţiei;
- *valori nenule*: valorile atributului (sau ale ansamblului de atribute) ce desemnează cheia primară sunt întotdeauna specificate, deci nenule şi, mai mult, niciun atribut din compoziţia cheii primare nu poate avea valori nule; această a treia condiţie se mai numeşte şi *restricţie a entităţii*.

Joe Celko inventariază patru proprietăţi dezirabile pentru o cheie ([Celko 1999], p. 247):

- familiaritate: valorile cheii trebuie să fie uşor de înţeles pentru utilizatori;
- stabilitate: valorile cheii nu trebuie să fie volatile;
- minimalitate;
- simplitate: sunt preferate cheile scurte şi simple.

Tot el atrage atenţia că cele patru proprietăţi pot intra, uneori, în conflict. O cheie stabilă poate să se dovedească complexă şi dificil de gestionat. Identificarea

cheii primare pentru o relaţie face parte (împreună cu stabilirea tabelelor prin gruparea atributelor, stabilirea celorlalte restricţii) din procesul de proiectare a bazei de date. Uneori, precum în cazul tabelei JUDEŢE, stabilirea cheii primare nu ridică probleme. Fiecare judeţ are un indicativ auto unic, deci atributul Jud candidează la postul de cheie a relaţiei. La fel se poate spune şi despre denumirea judeţului. Rezultă că relaţia JUDEŢE prezintă două chei candidate: Jud şi Judet. Alegerea unei dintre cheile candidat are în vedere criterii precum lungimea, uşurinţa în reţinere şi/sau editare. Fiind mai scurt, desemnăm atributul Jud drept cheie primară, situaţie în care Judet devine cheie alternativă.

Tabela CLIENŢI are cheia primară simplă - CodClient. CodClient reprezintă un număr unic asociat fiecărei firme căreia i-am făcut vânzări (clienţii noştri sunt exclusiv persoane juridice, adică firme). Există însă suficiente cazuri în care cheia primară este compusă din două, trei ş.a.m.d. sau, în cazuri extreme, din toate atributele relaţiei. Să analizăm o relaţie PERSONAL, care conţine date generale despre angajaţii firmei. Fiecare tuplu al relaţiei se referă la un angajat, atributele fiind: Nume, Prenume, DataNaşterii, Vechime, SalariuTarifar.

- Atributul Nume nu poate fi cheie, deoarece chiar şi într-o întreprindere de talie mijlocie, este posibil să existe doi angajaţi cu acelaşi nume.
- Dacă apariţia a două persoane cu nume identice este posibilă, atunci apariţia a două persoane cu acelaşi Prenume este probabilă.
- Niciunul dintre aceste atributele DataNaşterii, Vechime, SalariuTarifar nu poate fi „înzestrat" cu funcţia de identificator.

În acest caz, se încearcă gruparea a două, trei, patru ş.a.m.d. atribute, până când se obţine combinaţia care va permite diferenţierea clară a oricărei linii de toate celelalte. Combinaţia Nume + Adresă pare, la primele două vederi, a îndeplini „cerinţele" de identificator. Ar fi totuşi o problemă: dacă în aceeaşi firmă (organizaţie) lucrează împreună soţul şi soţia? Ambii au, de obicei, acelaşi nume de familie şi, tot de obicei, acelaşi domiciliu. Este adevărat, cazul ales nu este prea fericit. Dar este suficient pentru „compromiterea" combinaţiei.

Următoarea tentativă este grupul Nume + Prenume + Adresă, combinaţie neoperativă dacă în organizaţie lucrează tatăl şi un fiu (sau mama şi o fiică) care au aceleaşi nume şi prenume şi domiciliul comun. Ar rămâne de ales una dintre soluţiile (Nume + Prenume + Adresă + Vechime) sau (Nume + Prenume + Adresă + DataNaşterii).

Oricare din cele două combinaţii prezintă riscul violării restricţiei de entitate, deoarece este posibil ca, la preluarea unui angajat în bază, să nu i se cunoască adresa sau data naşterii, caz în care atributul respectiv ar avea valoarea NULL. Dificultăţile de identificare fără ambiguitate a angajaţilor au determinat firmele ca, la angajare, să aloce fiecărei persoane un număr unic, denumit Marcă. Prin adăugarea acestui atribut la cele existente, pentru relaţia PERSONAL problema cheii primare este rezolvată mult mai simplu. Actualmente, pentru cetăţenii români sarcina este simplificată şi prin utilizarea codului numeric personal (CNP), combinaţie de 13 cifre care prezintă avantajul că rămâne neschimbată pe tot parcursul vieţii persoanei.

Problema unicităţii a suscitat o serie de discuţii în modelul relaţional. Clasicii modelului, Codd şi Date, s-au pronunţat împotriva repetării tuplurilor în cadrul unei relaţii, cerinţă pe care standadele SQL nu au luat-o în considerare. Există şi situaţii reale în care repetarea liniilor este inevitabilă. David Beech a prezentat într-o lucrare înaintată Consiliului de Standardizare a bazelor de date ANSI X3H2, şi mai apoi în revista *Datamation* [Beech 1989], problema consacrată în literatura de specialitate drept „argumentul cutiilor cu mâncare pentru pisici" ([Celko 1999], p. 45): într-un supermagazin în care casele de marcat sunt legate la o bază de date, un client cumpără patru cutii de Whiskas[6] pentru pisici care costă, fiecare (cutie), 10 lei; pe bon apare de patru ori „cutii Whiskas". Autorul afirmă, pe bună dreptate, că la nivelul conceptual la care se înregistrează informaţiile, nu există nicio valoare prin care cele patru rânduri să fie diferenţiate. Astfel încât, pentru a respecta canoanele modelului relaţional, este necesară introducerea unei coloane suplimentare fără prea mare relevanţă, dar care să asigure unicitatea fiecărui tuplu. În acest sens, E.F. Codd a propus consiliului ANSI X3H2 o funcţie specială care să măsoare gradul de repetabilitate a valorii unuia sau mai multor atribute, funcţie care ar semăna întrucâtva cu funcţia COUNT() din SQL, folosită împreună cu GROUP BY ([Celko 1999], p. 49).

Pe lângă noţiunile cheie candidat, cheie primară, cheie alternativă, modelul relaţional utilizează şi sintagma *supercheie*. O supercheie poate fi definită ca un set de coloane ce îndeplineşte condiţia de cheie (unicitate), dar care conţine cel puţin un subset care este el însuşi cheie ([Celko 1999], p. 52). Cu alte cuvinte, dintre cele trei condiţii ale cheii primare, o supercheie o îndeplineşte numai pe prima (unicitate), fără a avea însă compoziţia minimală şi fără a pune problema restricţiei de entitate (valorile nule ale atributelor componente).

Domeniul unui atribut care este cheie primară într-o relaţie este denumit domeniu primar. Dacă într-o relaţie există mai multe combinaţii de atribute care conferă unicitate tuplului, acestea sunt denumite chei candidate. O cheie candidată care nu este identificator primar este referită drept *cheie alternativă.*

2.2.5. Restricţia referenţială

O bază de date relaţională este alcătuită din relaţii (tabele) aflate în legătură. Stabilirea legăturii se bazează pe mecanismul cheii *străine* (*externe*) şi, implicit, a restricţiei referenţiale. Figura 2.4 prezintă o legătură în care sunt implicate tabelele FACTURI şi CLIENŢI. Atributul CodClient joacă un rol de „agent de legătură" între tabelele CLIENŢI şi FACTURI. Pentru relaţia CLIENŢI, atributul CodClient este cheie *primară*, în timp ce în tabela FACTURI, CodClient reprezintă *coloana de referinţă* sau *cheia străină* (*externă*), deoarece numai pe baza valorilor sale se poate face legătura cu relaţia-părinte CLIENŢI.

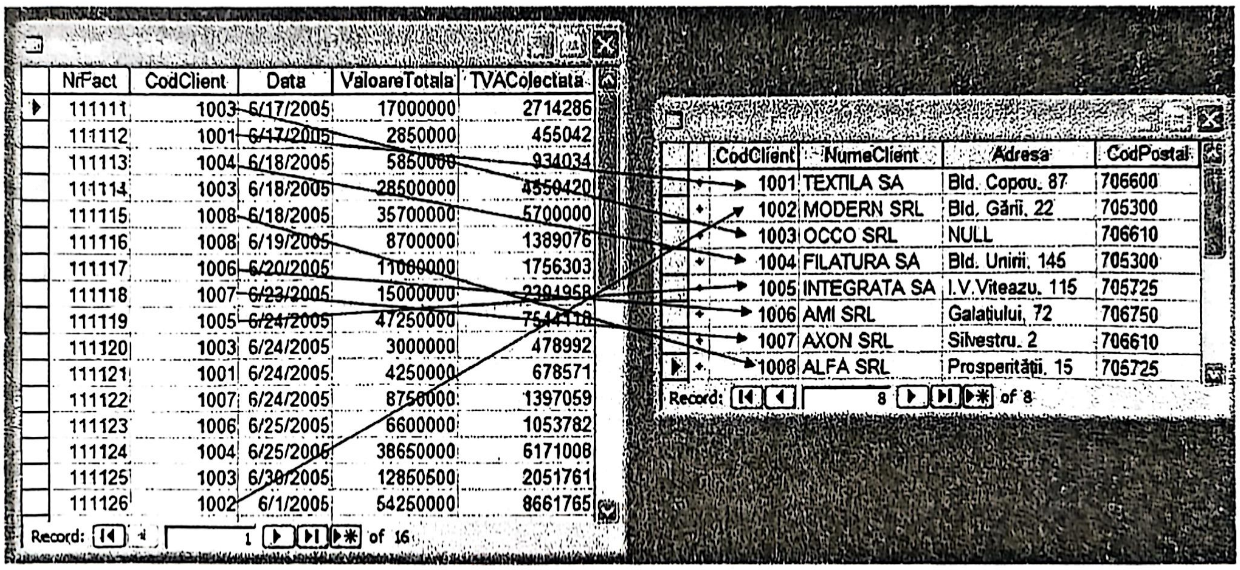
Cheile străine, denumite şi *chei externe* sau *coloane de referinţă*, sunt deci atribute sau combinaţii de atribute care pun în legătură linii (tupluri) din relaţii diferite.

6. Am rugămintea de a nu vă lăsa manipulaţi de reclame ieftine (ci numai de cele scumpe)!

Tabela în care atributul de legătură este cheie primară se numeşte *tabelă-părinte* (în cazul nostru, CLIENŢI), iar cealaltă *tabelă-copil*.

În legătură cu noţiunea de *cheie străină* apare conceptul de *restricţie referenţială*. O restricţie de integritate referenţială apare atunci când o relaţie face referinţă la o altă relaţie. C.J. Date defineşte restricţia de integritate referenţială astfel ([Date, 1986], p. 89): Fie D un domeniu primar şi R1 o relaţie ce prezintă un atribut A definit pe D - notate R1 (..., A:D, ...). În orice moment, orice valoare a lui A în R1 trebuie să fie:

a) ori nulă;
b) ori egală cu o valoare a lui V, unde V este cheie primară sau candidată într-un tuplu al unei relaţii oarecare R2 (R1 şi R2 nu trebuie să fie neapărat distincte), cheie primară definită pe acelaşi domeniu primar D.



NrFact	CodClient	Data	ValoareTotala	TVAColectata
111111	1003	6/17/2005	17000000	2714286
111112	1001	6/17/2005	2850000	455042
111113	1004	6/18/2005	5850000	934034
111114	1003	6/18/2005	28500000	4550420
111115	1008	6/18/2005	35700000	5700000
111116	1008	6/19/2005	8700000	1389076
111117	1006	6/20/2005	11000000	1756303
111118	1007	6/23/2005	15000000	2394958
111119	1005	6/24/2005	47250000	7544118
111120	1003	6/24/2005	3000000	478992
111121	1001	6/24/2005	4250000	678571
111122	1007	6/24/2005	8750000	1397059
111123	1006	6/25/2005	6600000	1053782
111124	1004	6/25/2005	38650000	6171008
111125	1003	6/30/2005	12850500	2051761
111126	1002	6/1/2005	54250000	8661765

Record: 1 of 16

CodClient	NumeClient	Adresa	CodPostal
1001	TEXTILA SA	Bld. Copou, 87	706600
1002	MODERN SRL	Bld. Gării, 22	705300
1003	OCCO SRL	NULL	706610
1004	FILATURA SA	Bld. Unirii, 145	705300
1005	INTEGRATA SA	I.V.Viteazu, 115	705725
1006	AMI SRL	Galaţiului, 72	706750
1007	AXON SRL	Silvestru, 2	706610
1008	ALFA SRL	Prosperităţii, 15	705725

Record: 8 of 8

Figura 2.4. O restricţie referenţială între FACTURI (tabelă-copil) şi CLIENŢI (părinte)

Într-o altă formulare, se iau în discuţie două tabele (relaţii) T1 şi T2. Ambele au atributul sau grupul de atribute notat CH, care pentru T1 este cheie primară, iar pentru T2 cheie străină. Dacă în T2 se interzice apariţia unor valori nenule ale CH, care nu există în niciun tuplu din T1, se spune că între T2 şi T1 s-a instituit o restricţie referenţială.

Instituirea restricţiei referenţiale între tabela CLIENŢI (părinte) şi FACTURI (copil) permite cunoaşterea, pentru fiecare factură, a denumirii şi adresei clientului căruia i-a fost adresată. Dacă în FACTURI ar exista vreo linie în care valoarea atributului CodClient ar fi, spre exemplu 9988, este clar că acea linie ar fi orfană (nu ar avea linie corespondentă în tabela părinte CLIENŢI).

Observaţii

- Pentru mulţi utilizatori şi profesionişti ai bazelor de date, denumirea de „relaţional" desemnează faptul că o bază de date este alcătuită din tabele puse în legătură (relaţie) prin intermediul cheilor externe şi primare. Aceasta este, de fapt, a doua accepţiune a termenului *baze de date relaţionale* (şi cea eronată),

prima, cea „clasică", având în vedere percepţia fiecărei linii dintr-o tabelă ca o relaţie între clase de valori.

- Majoritatea SGBD-urilor prezintă mecanisme de declarare şi gestionare automată a restricţiilor referenţiale, prin actualizări în cascadă şi interzicerea valorilor care ar încălca aceste restricţii.
- Respectarea restricţiilor referenţiale este una dintre cele mai complicate sarcini pentru dezvoltatorii de aplicaţii ce utilizează baze de date. Din acest punct de vedere, tentaţia este de a „sparge" baza de date în cât mai puţine tabele cu putinţă, altfel spus, de a avea relaţii cât mai „corpolente". Gradul de fragmentare al bazei ţine de *normalizarea* bazei de date, care, ca parte a procesului de proiectare a BD, se bazeză pe dependenţele funcţionale, multivaloare şi de joncţiune între atribute pe care, din fericire, nu le atacăm în această carte.

2.2.6. Restricţii-utilizator

Restricţiile utilizator mai sunt denumite şi *restricţii (reguli) de comportament* sau *restricţii (reguli) ale organizaţiei*. De obicei, iau forma unor *reguli la nivel de atribut* (*field validation rules*), *reguli la nivel de linie* (*record validation rules*) sau reguli complexe, interatribute/-linii/-tabele, implementabile prin *aserţiuni* (*assertions*), *reguli* (*rules*) sau *declanşatoare* (*triggers*).

Reguli la nivel de atribut

O restricţie la nivel de atribut poate preveni introducerea în baza de date a unor valori din alte intervale decât cele autorizate, în alte formate decât cele permise etc. O astfel de regulă înlocuieşte (sau completează, mai bine zis) o restricţie de domeniu. Forma clasică a unei restricţii la nivel de atribut este o expresie în care apar constante, funcţii-sistem şi, nu în ultimul rând, atributul respectiv. De obicei, expresia conţine, pe lângă constante şi funcţii-sistem (nici funcţiile-sistem nu sunt tolerate întotdeauna), numai atributul pentru care se defineşte restricţia. De exemplu, pentru atributul CodCl din tabela CLIENŢI, se poate declara restricţia: *CodCl BETWEEN 1001 AND 5678*. Sau în tabela STUDENŢI există atributul CicluStudii, care, în condiţiile (oarecum) noului sistem universitar european[7] (Bologna), poate avea valorile 1, 2 sau 3, iar regula poate fi formulată în mai multe variante:

- *CicluStudii > 0 AND CicluStudii < 4;*
- *CicluStudii >= 1 AND CicluStudii <= 3;*
- *CicluStudii BETWEEN 1 AND 3;*
- *CicluStudii IN (1, 2, 3).*

Numărul anilor de studii pe fiecare ciclu este însă uşor diferit. Ciclurile 1 (licenţă) şi 3 (doctorat) se întind pe durata a trei ani[8], în timp ce al doilea ciclu

7. Afirmaţia este discutabilă, ca şi noul „sistem", de altfel.
8. Regula se aplică în universităţile „clasice", universităţile de medicină şi politehnice având o altă structură.

(specializare & master) durează doi ani. Așa că, pentru AnStudii regula este similară atributului CicluStudii:

- *AnStudii >= 1 AND AnStudii <= 3;*
- *AnStudii BETWEEN 1 AND 3;*
- *AnStudii IN (1, 2, 3).*

Dacă, însă, dorim să fim mai exacți și să ținem cont că valoarea maximă a anului de studii pentru al doilea ciclu este 2, putem imagina o regulă mai complexă: *CASE WHEN CicluStudii IN (1,3) THEN AnStudii BETWEEN 1 AND 3 ELSE AnStudii BETWEEN 1 AND 2 END*. Ei bine, această regulă (expresie) conține două atribute, nefiind una la nivel de atribut, ci *la nivel de înregistrare.*

La orice editare a atributului cu pricina (declanșată la inserarea unei linii în tabela din care face parte sau la modificarea valorii sale), expresia este evaluată și, dacă rezultatul evaluării are valoarea logică adevărat (TRUE), atunci inserarea/modificarea este permisă, iar dacă rezultatul este FALSE, atunci inserarea/modificarea este blocată. Astfel, dacă se inserează o înregistrare în tabela STUDENȚI, ce corespunde unui student proaspăt înmatriculat (în urma admiterii sau transferului), iar valoarea specificată pentru atributul CicluStudii este 4, atunci expresia *CicluStudii BETWEEN 1 AND 3* are valoarea logică *FALSE*, iar SGBD-ul nu va tolera inserarea. Ulterior, o altă comandă de inserare în care valoarea atributului ar fi cuprinsă între 1 și 3, ar putea fi acceptată, în caz că nicio altă restricție nu ar fi încălcată.

Reguli la nivel de înregistrare

După cum am văzut mai sus, expresia care definește o restricție la nivel de înregistrare conține două sau mai multe atribute: *CASE WHEN CicluStudii IN (1,3) THEN CASE WHEN AnStudii BETWEEN 1 AND 3 THEN TRUE ELSE FALSE END ELSE CASE WHEN AnStudii BETWEEN 1 AND 2 THEN TRUE ELSE FALSE END END*[9]. Restricția la nivel de înregistrare este evaluată la inserarea unei linii sau la modificarea valorii a măcar unui atribut într-o înregistrare din tabelă. Dacă, teoretic (și, uneori, practic) restricția unui atribut se verifică imediat după modificarea valorii acestuia, restricția la nivel de înregistrare se verifică după modificarea tuturor atributelor de pe linia respectivă. În practică, este greu de delimitat aceste momente, pentru că deseori se produc aproape simultan.

Tabela FACTURI ar putea conține, printre altele, două atribute „valorice", unul pentru păstrarea valorii totale (inclusiv TVA) a facturii respective și un altul care stochează cuantumul TVA colectat pentru factură. Taxa pe valoarea adăugată pentru majoritatea produselor și serviciilor comercializate la noi în țară reprezintă o cotă de 19%. Există, însă, și produse la care cota poate fi 9% sau chiar 0%. De aceea, TVA colectată pentru o factură este mai mică sau cel mult egală decât/cu 19% din valoarea fără TVA. Să încercăm o formulă: *ValoareaTotală = ValoareaFărăTVA + TVA*. Dacă factura conține numai produse cu 19%:

9. Este o altă variantă (care se vrea mai impresionantă) a restricției prin care anul de studii se sincronizează cu ciclul de studii.

*ValoareaTotală = ValoareaFărăTVA + 0.19 * ValoareaFărăTVA* sau *ValoareaTotală = 1.19 * ValoareaFărăTVA*. Dar tabela noastră are doar atributele ValoareTotală şi TVAColectată, aşa că înlocuim ValoareaFărăTVA prin diferenţa celorlalte două. Scriem: *ValoareTotală = 1.19 * (ValoareTotală - TVAColectată)*, iar după calcule de matematică superioară, *TVAColectată * 1.19 = 0.19 * ValoareTotală*, altfel spus *TVAColectată = ValoareTotală * 0.19 / 1.19*[10]. Prin urmare, expresia restricţiei la nivel de înregistrare este: *TVAColectată BETWEEN 0 AND ValoareTotală * 0.19 / 1.19*[11].

Să ne imaginăm şi alte asemenea restricţii:

O tabelă ÎNCASĂRI (vezi paragraful 2.3.2), ar putea conţine un atribut DataÎncasării, care să indice ziua în care ne-au intrat banii în cont (sau casierie) şi un altul, DatăDocument. Ideea este că un client ne poate plăti o sumă printr-un ordin de plată pe care îl întocmeşte pe 9 septembrie 2008 (aceasta este valoarea atributului DatăDocument), iar banii să ajungă în contul nostru bancar pe 12 septembrie 2008 (aceasta este valoarea atributului DataÎncasării). Regula s-ar referi la precedenţa celor două valori: *CASE WHEN DataÎncasării IS NOT NULL THEN DataÎncasării >= DatăDocument END*.

Într-o altă tabelă - GĂRZI - din baza de date a unui spital de urgenţă, ar putea exista un atribut pentru evidenţierea momentului începutului (DatăOrăÎnceputGardă) şi sfârşitului (DatăOrăSfârşitGardă) unei gărzi. Restricţia la nivel de înregistrare ar privi tot precedenţa celor două valori: *DatăOrăÎnceputGardă < DatăOrăSfârşitGardă*.

Cele două tipuri de restricţii pot lucra în tandem. Astfel, pentru tabela STUDENŢI putem declara:

- o restricţie pentru atributul CicluStudii: *CicluStudii IN (1, 2, 3);*
- o restricţie pentru atributul AnStudii: *AnStudii IN (1, 2, 3);*
- o regulă la nivel de înregistrare care ar avea o expresie mult simplicată: *CASE WHEN CicluStudii = 2 THEN AnStudii <= 2 END*.

Reguli complexe

Există şi alte tipuri de restricţii a căror validare presupune verificarea valorilor (unuia sau mai multor atribute) de pe două sau mai multe linii (înregistrări) ale aceleiaşi tabele sau chiar aflate în tabele diferite. Iată câteva exemple mai mult sau mai puţin realiste:

- O factură poate avea maximum 30 de linii (produse vândute).
- Se interzice emiterea unei noi facturi (o nouă vânzare), dacă datoriile firmei-client sunt mai mari de 20.000 de lei (iar directorul acesteia nu este membru în partidul/partidele de guvernământ).
- Pentru o bibliotecă universitară, numărul maxim de cărţi împrumutate, la orice moment dat, depinde de categoria cititorului: 5 pentru studenţii din ciclul 1, 7 pentru studenţii din ciclurile 2 şi 3 şi 10 pentru profesori.

10. Această expresie reprezintă chintesenţa desăvârşirii mele matematice.
11. În economia lucrării, după acest paragraf, pentru autor a urmat o pauză de jumătate de zi ca urmare a epuizării intelectuale.

Modalităţile prin care aceste restricţii pot fi declarate/implementate sunt destul de eterogene în standardele SQL şi în practică (SGBD-uri): aserţiuni, reguli şi, probabil, cele mai frecvente, declanşatoarele (*trigger*-ele) pe care le vom discuta în detaliu spre finalul cărţii.

2.3. Schema şi conţinutul unei baze de date relaţionale

Aşa cum era prezentat în primul capitol, există două aspecte complementare de abordare a bazelor de date: *schema* (structura) şi *conţinutul* (instanţierea, extensia). Pentru o bază de date relaţională (BDR), *conţinutul* unei relaţii este reprezentat de ansamblul tuplurilor care o alcătuiesc la un moment dat. *Schema* unei baze de date conţine denumiri ale tabelelor, numele, tipul şi lungimea atributelor, restricţii de unicitate, de nenulitate, restricţii la nivel de atribut şi de linie şi alte eventuale tipuri de restricţii de comportament, precum şi restricţii referenţiale. La acestea se adaugă drepturile utilizatorilor, definiţiile şi restricţiile tabelelor virtuale, indexurilor etc. Foarte importante în lumea profesioniştilor dezvoltării de aplicaţii cu bazele de date sunt procedurile stocate - programe de forma funcţiilor, procedurilor, pachetelor şi declanşatoarelor (*trigger*-elor) - care, după cum le spune şi numele, sunt memorate în schema bazei de date şi sunt parte integrantă din aceasta. Mai sunt şi alte ingrediente ale unei scheme de BDR, însă le vom prezenta pe parcurs.

2.3.1. Reprezentarea schemei unei baze de date relaţionale

Schema unei BDR poate fi (şi chiar trebuie) reprezentată grafic. Există o mulţime de reprezentări, mai sugestive sau mai criptice, fiind greu de făcut o clasificare din punctul de vedere al lizibilităţii/comprehensibilităţii lor. Oricum, ierahizarea ar fi, din start, contaminată de preferinţele şi experienţa autorului.

Prima ediţie a cărţii a recurs la un formalism „neutru", bazat pe următoarele reguli simpliste[12]:

- O tabelă se reprezintă pe două linii, pe prima fiind scris numai numele relaţiei iar pe cea de-a doua numele atributelor; ordinea prezentării atributelor nu are importanţă, însă, pentru lizibilitate, coloana sau coloanele care desemnează cheia primară se dispun succesiv, la început (stânga); tot alăturat se dispun şi atributele care constituie o cheie străină.
- Numele coloanelor ce alcătuiesc cheia primară se subliniază cu o linie continuă, iar cele care alcătuiesc cheile alternative se subliniază cu linie punctată.
- Numele coloanelor facultative (atributele ce pot avea valori NULL) sunt scrise între paranteze.
- Dacă o cheie străină este alcătuită din mai multe coloane, se poate utiliza acolada pentru a le grupa.

12. Preluare din [Hainaut 1994], pp. 24-25.

- O restricţie referenţială este reprezentată printr-o săgeată care pleacă din cheia externă (sau de la vârful acoladei, în cazul unei chei străine compuse) şi are vârful înfipt în cheia primară a tabelei-părinte.

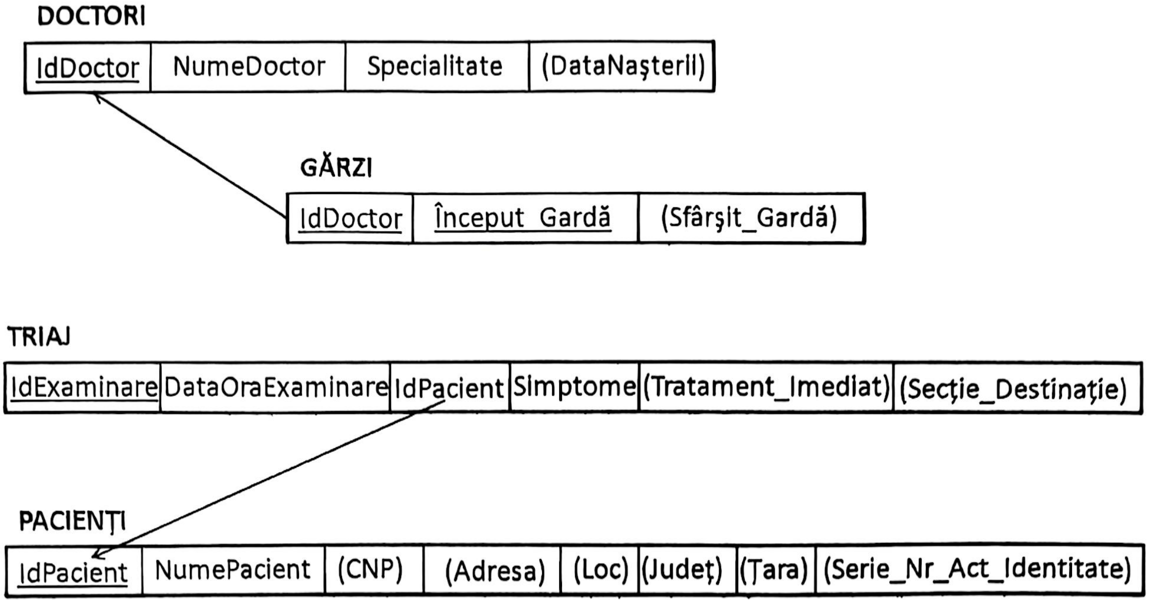


Figura 2.5. Una dintre reprezentările BD TRIAJ

Astfel, o bază de date destul de simplistă dedicată unui spital de urgenţă, BD denumită TRIAJ ar putea avea schema reprezentată ca în figura 2.5. Faptul că există o tabelă cu acelaşi nume ca al bazei de date nu e grozav, dar nici toxic.

Un formalism frecvent printre nonprofesionişti este cel din Access - vezi figura 2.6. Principalul avantaj este că, datorită modului de dispunere a atributelor în tabele, reprezentarea relaţiilor cu multe atribute nu ridică probleme precum formalismul anterior. Există însă şi câteva dezavantaje majore:

- Nu se pot vizualiza cheile alternative (alte atribute/grupuri de atribute cu valori unice în tabela respectivă).
- Nu se pot deosebi atributele cu valori obligatorii (NOT NULL) de cele pentru care se pot accepta valori nule.
- Restricţiile referenţiale trebuie ghicite după numele atributelor, întrucât linia care leagă tabela-copil de tabela-părinte nu este trasată (decât accidental) în dreptul atributelor cheie externă şi cheie primară.
- Pentru Access, conectarea unei tabele-părinte cu una copil nu înseamnă automat şi că restricţia referenţială a fost instituită.

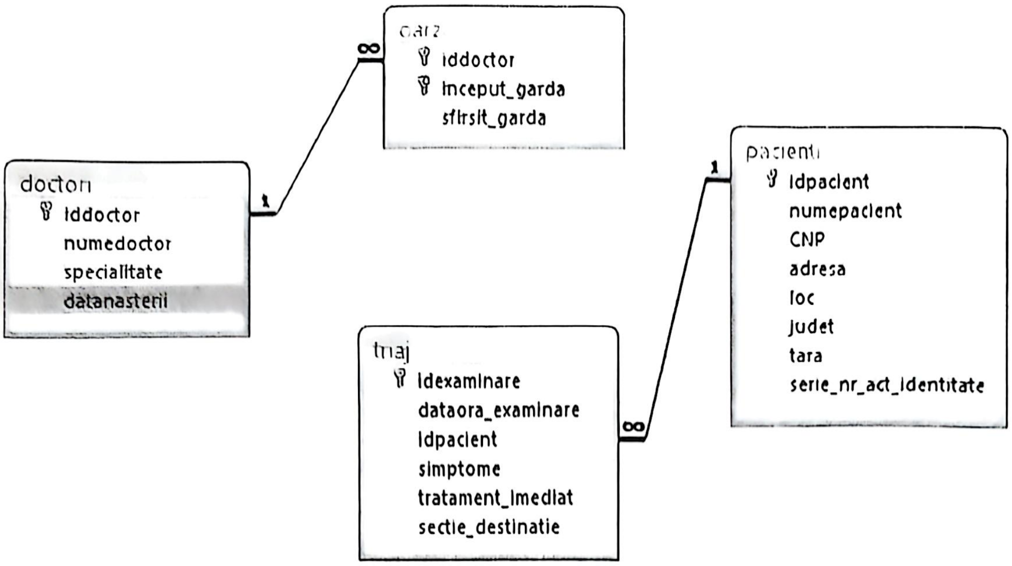


Figura 2.6. Reprezentarea din Access a mini-BD TRIAJ

În această ediţie nu vom lucra (prea mult) cu MS Access. Dintre cele patru SGBD-uri alese (vezi paragraful 2.4), SQL Server (produs de Microsoft) pune la dispoziţie un mecanism de reprezentare la fel de simplu precum cel din Access - vezi figura 2.7, formalismul „moştenind", de fapt, atât avantajele, cât şi dezavantajele celui din Access.

2.3.2. Scheme folosite pentru ilustrarea interogărilor în algebra relaţională şi SQL

Pe parcursul următoarelor capitole, ne vom folosi, în principal, de două baze de date, dintre care prima este, mai degrabă, o mini sau microbază de date.

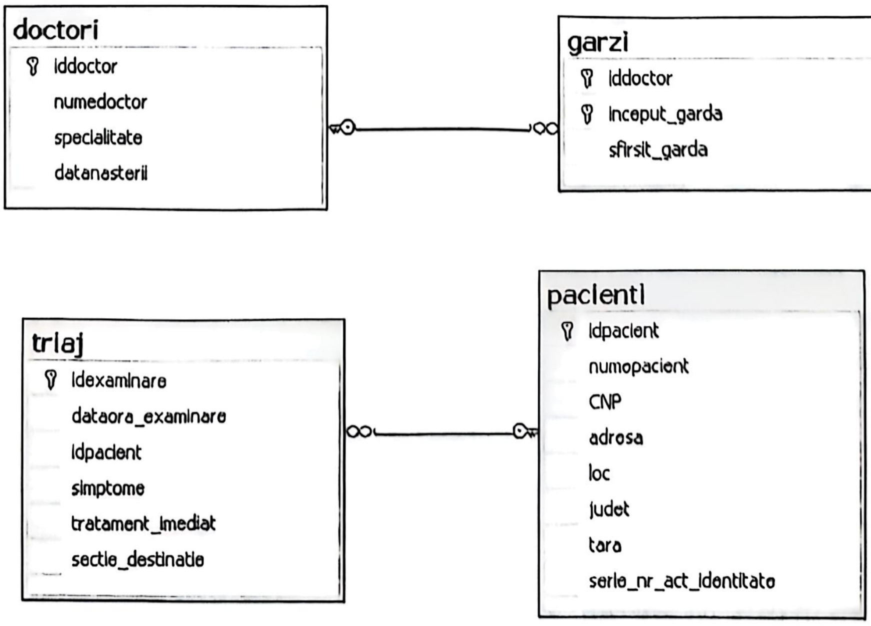


Figura 2.7. Reprezentarea în SQL Server a mini-BD TRIAJ

Baza de date TRIAJ

Deşi nu este cea mai importantă în economia lucrării, începem prin a detalia BD TRIAJ. Aceasta conţine patru tabele, DOCTORI, GĂRZI, PACIENŢI şi TRIAJ care stochează informaţii generale despre cazurile recepţionate la camera de gardă în decursul unei perioade de câţiva ani.

Tabela DOCTORI păstrează, pe fiecare linie (înregistrare), date generale despre fiecare medic al clinicii (spitalului), atributele sale fiind:

- IdDoctor - un număr întreg care identifică fiecare doctor (un fel de marcă sau cod);
- NumeDoctor - şir de caractere al cărui rol este evident;
- Specialitate - şir de caractere ce identifică domeniul în care este pregătit doctorul (boli interne, chirurgie cardiovasculară etc.);
- DataNaşterii - o dată calendaristică asupra căreia nu mai are rost să insistăm.

Cheia primară este IdDoctor, iar pentru ultimele două atribute se pot accepta valori nule.

Tabela GĂRZI conţine programarea medicilor în gărzile clinicii. Fiecare linie se referă la un medic care face o gardă. Atribute:

- IdDoctor - un număr întreg care identifică medicul de gardă;
- Început_Gardă - o combinaţie dată + oră (inclusiv minut, secundă), să-i zicem moment în care medicul intră în gardă;
- Sfârşit_Gardă - momentul (ziua şi ora) la care medicul a ieşit sau va ieşi din gardă.

Cheia primară a tabelei este combinaţia (IdDoctor, Început_Gardă), ceea ce înseamnă, printre altele, că, în spitalul cu pricina, într-o aceeaşi gardă pot fi programaţi doi sau mai mulţi medici (în perioadele aglomerate). Fiind componente ale cheii primare, aceste două atribute nu pot accepta valori nule. Ultimul însă poate avea valori NULL, ceea ce ar însemna că este vorba despre garda în curs, cea care nu s-a terminat.

IdDoctor este cheie străină în tabela GĂRZI, cheia primară corespondentă fiind atributul cu acelaşi nume din tabela DOCTORI.

Tabela PACIENŢI „oferă" câte o linie pentru fiecare bolnav, fericit sau nefericit, care a trecut pragul spitalului (şi a fost înregistrat). Atribute:

- IdPacient - un număr întreg care identifică fiecare persoană/pacient;
- NumePacient - şir de caractere cu funcţionalitate evidentă;
- CNP - codul numeric personal al pacientului;
- Adresa - şir de caractere ce indică domiciliul (din cartea de identitate) pacientului;
- Loc - localitatea în care îşi are domiciliul pacientul (este separată de adresă);
- Judeţ - judeţul în se care se găseşte localitatea;
- Ţară - ţara în care îşi are domiciliul pacientul;
- Serie_Nr_Act_Identitate - seria şi numărul actului de identitate al pacientului.

Cheia primară a tabelei este IdPacient. E interesant că acesta este singurul atribut al tabelei care nu poate accepta valori nule, ceea ce pare destul de curios. Motivul este acela că la clinică pot sosi victime neînsoţite ale unor accidente rutiere care nu au asupra lor niciun act de identitate şi, până în momentul în care devin conştiente şi li se recuperează documentele, trebuie preluate în baza de date fără a se şti mai nimic despre ele.

Tabela TRIAJ conţine, pe fiecare dintre linii, un „caz", adică înregistrarea sosirii unei persoane în stare mai bună sau mai rea. În decursul anilor, o persoană care se regăseşte în tabela PACIENŢI pe o singură linie/înregistrare poate avea mai multe linii din TRIAJ corespondente. Astfel, un ghinionist poate a avut pe 13 mai 2005 o criză renală, pe 13 ianuarie 2007 a alunecat pe gheaţă şi şi-a rupt un picior, iar pe 13 iunie 2008 a avut un accident rutier în care şi-a rupt o coastă. Toate cele trei „bucurii" s-au petrecut în preajma spitalului, care îl poate considera client fidel. Atribute:

- IdExaminare - un număr întreg care identifică orice sosire a unui pacient în camera de gardă, sosire care se transformă în „caz";
- DataOra_Examinare - momentul (data/ora) sosirii/examinării pacientului;
- IdPacient - numărul întreg care identifică pacientul;
- Simptome - şir de caractere în care medicul de gardă descrie cum se prezenta pacientul în momentul sosirii;
- Tratament_Imediat - medicamente, masaje cardiace şi alte operaţiuni pe care medicul de gardă le-a administrat/aplicat pacientului, ţinând seama de simptome;
- Secţie_Destinaţie - şir de caractere care indică eventuala secţie în care a fost transferat pacientul, după administrarea tratamentului imediat.

Cheia primară a tabelei este IdExaminare. Acest atribut, DataOra_Examinare şi IdPacient nu pot primi valori nule. IdPacient este cheie străină (tabela-părinte fiind PACIENŢI).

Baza de date VÂNZĂRI

În capitolele următoare vom utiliza cu precădere o bază de date „martor", denumită VÂNZĂRI, a cărei schemă simplificată este cea din figura 2.8.

Tabela JUDEŢE conţine câte o linie pentru fiecare judeţ în care sunt clienţi. Atribute:

- Jud - indicativul auto al judeţului (alcătuit din două litere, majuscule);
- Judeţ - denumirea judeţului;

Regiune - regiunea istorică (provincia) din care face parte judeţul.

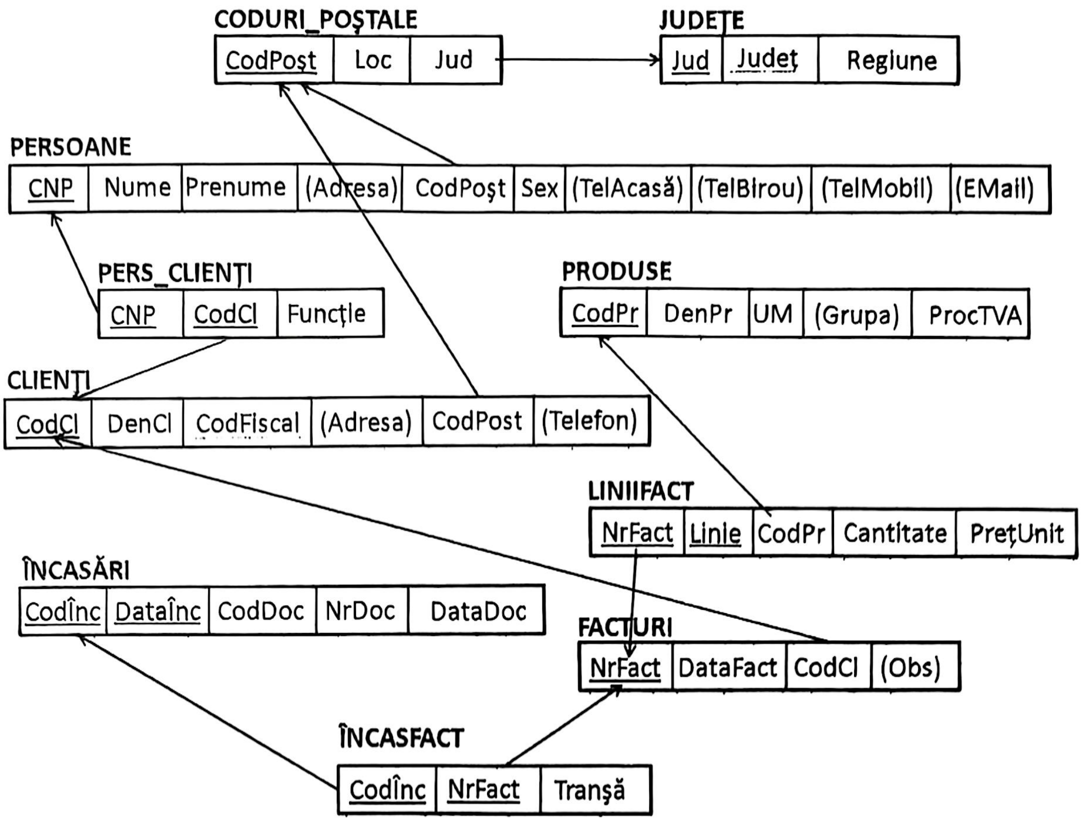


Figura 2.8. Schema simplificată a bazei de date VÂNZĂRI

Cheia primară este atributul Jud, iar atributul Judeţ este cheie alternativă. Niciunul dintre atribute nu acceptă valori nule. Restricţii-utilizator:

- Valoarea atributului Jud trebuie să fie alcătuită numai din majuscule (eventual, un spaţiu, pentru Bucureşti).
- Fiecare cuvânt din valoarea atributului Judeţ începe cu majusculă; restul literelor trebuie să fie mici (minuscule).
- Valoarea atributului Regiune începe cu o majusculă; restul literelor trebuie să fie mici.
- Regiune poate avea numai una dintre valorile: Banat, Dobrogea, Muntenia, Oltenia, Transilvania, Moldova.

Tabela CODURI_POŞTALE conţine câte o linie pentru fiecare cod poştal dintr-un oraş sau comună:

- CodPoşt – codul poştal;
- Loc – denumirea comunei/oraşului;
- Jud – indicativul auto al judeţului în care se află oraşul/comuna.

Cheia primară este atributul CodPoşt. Atributul Jud este cheie străină, tabela-părinte fiind JUDEŢE. Nici CodPoşt, nici Loc nu trebuie să accepte valori NULL. Restricţii-utilizator:

- Valoarea lui Jud este alcătuită numai din majuscule (eventual, un spaţiu, pentru Bucureşti).
- Fiecare cuvânt din Loc începe cu majusculă; restul literelor sunt mici.

Tabela CLIENȚI păstrează date generale ale clienților firmei pentru care s-a constituit baza de date (o linie - un client). Atribute:

- CodCl - codul clientului;
- DenCl - denumirea clientului (persoană juridică);
- CodFiscal - codul fiscal,
- Adresa - adresa sediului firmei-client;
- CodPoşt - codul poştal al comunei sau oraşului;
- Telefon - telefonul (principal) al clientului.

Cheia primară este atributul CodCl, iar CodFiscal este cheie alternativă, întrucât clienții sunt exclusiv persoane juridice (firme). Atributul CodPoşt este cheie străină, tabela-părinte fiind CODURI_POŞTALE. DenCl şi Adresa încep cu majuscule. Numai Adresa şi Telefon pot avea valori NULL. De asemenea, valoarea minimă acceptată pentru atributul CodCl este 1001.

Tabela PERSOANE stochează câteva date despre persoanele-cheie din firmele-client: directori generali, directori financiari, şefii departamentelor comerciale (aprovizionare şi/sau vânzări) etc. Probabil vi se pare ciudat, dar ceea ce intenționăm să facem cu această tabelă (şi cu următoarea) este să sprijinim fidelizarea clientului. Nu costă (mai) nimic dacă de Sfântul Ioan se trimite, din partea firmei noastre, câte o felicitare tuturor Ionilor şi Ioanelor. Iar pentru ca lucrurile să fie şi mai bine puse la punct, ar fi trebuit să preluăm şi informații precum: data naşterii, starea civilă, numele şi vârsta copiilor, pasiuni în materie de muzică, arte pastice, literatură, sport etc. Schema luată în considerare s-a oprit la următoarele atribute:

- CNP - codul numeric al persoanei;
- Nume;
- Prenume;
- Adresa - domiciliul stabil (din cartea de identitate);
- Sex;
- CodPoşt - codul poştal din adresă;
- TelAcasă - numărul de telefon de acasă;
- TelBirou - numărul de telefon (fix) de la birou;
- TelMobil - telefonul mobil;
- Email - adresa de e-mail.

Deşi este un număr compus din 13 poziții, este recomandabil ca CNP-ul să fie declarat de tip şir de caractere, deoarece în firmă există rezidenți din alte țări care au poziții de vârf în companii româneşti (sau reprezentanțe ale unor companii străine). Or, codul de identificare din paşaport poate fi alcătuit din litere şi cifre.

Cheia primară este atributul CNP (să presupunem că firma le poate obține, deşi e puțin plauzibil). Fiind vorba despre persoane „exterioare" companiei, un atribut precum Marca este mai puțin indicat. Atributul CodPoşt este cheie străină, tabela-părinte fiind CODURI_POŞTALE. Alte restricții:

- Fiecare cuvânt din Nume şi Prenume începe cu majusculă; restul literelor sunt mici.

- Adresa începe cu majusculă.
- Atributul Sex poate lua numai două valori: M de la masculin şi F de la Feminin.

Cu ajutorul tabelei PERSCLIENŢI stocăm funcţia deţinută de fiecare persoană la unul (sau mai multe, deşi aceste cazuri sunt rare) firme-client. O linie din această tabelă reflectă o funcţie deţinută de o persoană într-o firmă. O persoană poate avea mai multe funcţii, chiar la aceeaşi firmă (cumul de funcţii). Cele trei atribute sunt:
- CNP - codul numeric personal al persoanei care deţine funcţia;
- CodCl - firma la care persoana deţine funcţia;
- Funcţie - funcţia deţinută.

Cheia primară este compusă din toate cele trei atribute, (CNP, CodCl, Funcţie), deoarece, pe de o parte, o persoană poate cumula două sau mai multe funcţii la aceeaşi firmă, iar, pe de altă parte, o persoană poate, în unele cazuri să lucreze la două sau mai multe firme. CNP şi CodCl sunt chei străine, tabelele-părinte fiind PERSOANE, respectiv CLIENŢI. Atributul Funcţie începe cu o majusculă, restul literelor fiind mici.

Tabela PRODUSE reprezintă nomenclatorul de produse şi servicii pe care le comercializează firma (produsele sunt obţinute prin manufacturare sau revânzare). Atribute:
- CodPr - codul produsului;
- DenPr - denumirea;
- UM - unitatea de măsură a produsului;
- Grupa - grupa de mărfuri (sortimente) în care se încadrează; acest atribut este important pentru analiza vânzărilor;
- ProcTVA - procentul TVA care se aplică la preţul de vânzare (preţ care este fără TVA); pare de prisos în condiţiile actuale, dacă toate produsele vândute de firma noastră ar avea un singur procent, 19%; există însă şi produse la vânzarea cărora se aplică doar 9% şi, în plus, nimeni nu poate garanta cum vor gândi promoţiile viitoare de guvernanţi „relaxarea fiscală".

Cheia primară este atributul CodPr. DenPr şi Grupa încep cu majusculă, restul literelor din valoarea acestor atribute fiind mici.

Tabela pare una „inofensivă", dar de modul său de organizare depinde rezolvarea unor probleme sensibile. Este important de stabilit regimul în care se va lucra cu produsele care se comercializează sub mai multe forme (cu unităţi de măsură diferite). Atributul DenPr nu a fost declarat cheie alternativă. De ce? Să luăm un exemplu absolut la întâmplare: o firmă de comerţ angro vinde, printre altele, produsul Vodcă X la cutii de 1 litru şi la peturi (sticle de plastic) de 250 ml.

Deşi este acelaşi produs, pentru o evidenţă corectă a vânzărilor, este necesar ca în tabela PRODUSE să existe două linii afectate vodcii Scandic, fie ca în figura 2.9, fie ca în figura 2.10.

PRODUSE

CodPr	DenPr	UM	Grupa	ProcTVA
...				
1234	Vodca X - cutie 1 l	buc	Spirtoase	0.19
...				
1273	Vodca X - pet 250 ml	buc	Spirtoase	0.19
...				

varianta 1:

Cheie primară: CodPr
Cheie alternativă: DenPr

Figura 2.9. Specificarea, în valoarea DenPr, a tipului cutiei

PRODUSE

CodPr	DenPr	UM	Grupa	ProcTVA
...				
1234	Vodca X	cutie 1 l	Spirtoase	0.19
...				
1273	Vodca X	pet 250 ml	Spirtoase	0.19
...				

varianta 2:

Cheie primară: CodPr
Cheie alternativă: (DenPr, UM)

Figura 2.10. Diferențierea (în afară de cod) numai prin UM

Fiecare variantă are avantajele şi dezavantajele sale.

Tabela FACTURI conține câte o linie pentru fiecare factură emisă, ce reflectă o vânzare (către un client). Atribute:

- NrFact - numărul facturii;
- DataFact - data întocmirii facturii;
- CodCl - codul clientului căruia i s-au vândut produsele/serviciile consemnate în factură;
- Obs - observații; e folosit relativ rar, pentru a introduce eventuale detalii sau probleme care au apărut în legătură cu o factură.

Cheia primară este atributul NrFact. Explicația este una simplă: gestionarea facturilor emise este, conform legii, strictă. Nu pot exista două facturi (care reflectă două vânzări) cu un acelaşi număr[13]. CodCl este cheie străină (tabela-părinte este CLIENȚI). Ca restricție-utilizator suplimentară, să zicem că interzicem preluarea

13. În ultimii ani, legislația s-a mai domolit în privința numerotării facturilor. Companiile pot refolosi numere de facturi în fiecare an (prima factură din orice an poate avea, spre exemplu, numărul 100001), însă noi vom menține restricția de unicitate a valorilor NrFact în tabela FACTURI.

facturilor întocmite înainte de 1 ianuarie 2007 sau după 31 decembrie 2015 (*DataFact >= DATE'2007-01-01' AND DataFact <= DATE'2015-12-31'*).

Tabela LINIIFACT „detaliază" tabela precedentă. Un tuplu se referă la un produs/serviciu vândut şi consemnat într-o factură emisă. Pentru fiecare factură vor fi atâtea linii câte produse/servicii au fost consemnate la vânzarea respectivă. Atribute:

- NrFact - numărul facturii;
- Linie - numărul liniei din factura respectivă (1, 2, 3, ...);
- CodPr - codul produsului/serviciului vândut;
- Cantitate - cantitatea vândută;
- PretUnit - preţul unitar (fără TVA) la care s-a făcut vânzarea.

Cheia primară este combinaţia (NrFact, Linie). NrFact şi CodPr sunt chei străine. Pentru a determina valoarea de încasat a unei facturi (inclusiv TVA), la valoarea fără TVA pentru fiecare linie (obţinută prin produsul *Cantitate * PretUnit*) trebuie adăugată TVA colectată, obţinută prin aplicarea procentului de TVA al produsului/serviciului (atributul ProcTVA din PRODUSE) la valoarea fără TVA.

Tabela ÎNCASĂRI reprezintă „nomenclatorul" încasărilor. Printr-o încasare, un client îşi stinge una sau mai multe obligaţii de plată, adică achită una sau mai multe facturi. Documentul primar pe baza căruia se consemnează încasarea poate fi ordinul de plată, cecul, chitanţa etc. Atributele tabelei sunt:

- CodInc - codul încasării - un număr intern, util pentru a diferenţia o încasare de celelalte;
- DataInc - data încasării, data la care banii au intrat în contul sau casieria firmei;
- CodDoc - codul documentului justificativ al încasării: OP - ordin de plată, CHIT - chitanţă, CEC - filă cec etc.;
- NrDoc - numărul documentului justificativ;
- DataDoc - data la care a fost întocmit documentul justificativ; din momentul întocmirii documentului de plată până la data la care banii ajung efectiv în cont/casierie trec câteva zile datorită circuitului documentelor între firme şi bănci sau intervalul specificat în biletul la ordin sau în cambie.

Cheia primară este atributul CodÎnc. Ca restricţii utilizator pot fi instituite:

- data încasării nu o poate preceda pe cea a întocmirii documentului (*DataÎnc >= DataDataDoc*);
- data documentului de încasare şi data încasării efective trebuie să fie în intervalul de 1 ianuarie 2007 - 31 decembrie 2015 (*DataDoc >= DATE'2007-01-01' AND DataDoc <= DATE'2015-12-31'*);
- codul documentului se scrie numai cu majuscule.

Tabela ÎNCASFACT detaliază tabela precedentă şi indică ce facturi (tranşe din facturi) sunt achitate prin fiecare încasare. Un client poate plăti mai multe facturi odată (printr-un singur document). Pe de altă parte, orice factură poate fi plătită

într-una sau mai multe tranșe, în funcție de banii de care dispune clientul la momentul întocmirii documentului de plată. Atributele tabelei sunt:

- CodÎnc – codul încasării;
- NrFact – factura pentru care se încasează valoarea integrală sau numai o tranșă;
- Tranșă – tranșa din factură (sau întreaga valoare) care se încasează prin documentul primar ce stă la baza încasării.

Cheia primară este combinația (CodÎnc, NrFact), deoarece la o încasare se pot achita mai multe facturi, iar pe de altă parte, o factură poate fi plătită în mai multe tranșe. CodÎnc și NrFact sunt chei străine. Respectarea restricției de entitate presupune ca NrFact să nu poată avea valori nule, ceea ce atrage după sine un serios neajuns al acestei tabele – imposibilitatea de a prelua încasările în avans (înaintea întocmirii facturilor).

2.3.3. Alte obiecte din schema unei baze de date relaționale

Pe lângă tabele, atribute și restricții, dicționarul de date al unei BDR conține multe alte tipuri de obiecte, dintre care vom zăbovi foarte puțin asupra tabelelor temporare și virtuale (*view*-uri), procedurilor stocate și secvențelor. Cele mai „vechi", în ordinea apariției lor în teoria relațională, sunt tabelele persistente (de bază, „clasice") și tabelele virtuale. Unele, precum procedurile stocate și secvențele, nu fac parte din modelul relațional și au apărut în SGBD-uri din rațiuni practice, fiind încorporate ulterior în stardardul SQL.

Tabele temporare

O caracteristică fundamentală a oricărei BD este persistența. Tabelele prezentate până acum sunt cele care asigură această caracteristică, întrucât, după crearea lor (și, implicit, stocarea schemei lor în dicționarul de date), informațiile introduse și modificate sunt păstrate pe o durată nedeterminată, până la ștergerea lor explicită sau până la vreun nedorit dezastru hardware sau software.

Mai mult, în condițiile în care la baza de date are acces un mare număr de utilizatori, modificările efectuate de un utilizator (și confirmate prin încheierea eventualei tranzacții) în conținutul unei tabele sunt vizibile celorlalți utilizatori. Există însă și posibilitatea de a declara tabele a căror schemă să fie stocată în dicționar, dar pentru care conținutul să fie volatil. Astfel, pot fi create tabele al căror conținut să fie vizibil doar *sesiunii* de lucru care face apel la tabela respectivă, pe perioada întregii sesiuni. Acestea sunt tabelele *temporare globale*. Structura tabelelor *temporare locale* este, de asemenea, păstrată în dicționarul de date, însă vizibilitatea conținutului lor este limitată doar la *un singur modul/bloc* apelant în cadrul sesiunii. La ce ar putea folosi tabelele al căror conținut ar fi așa de efemer? Să luăm un exemplu: dorim să analizăm structura vânzărilor pe o regiune din care face parte utilizatorul (sau o regiune aleasă), după mai multe criterii, pe ultimele 12 luni. Datele fiind „istorice", iar BD folosită de mulți utilizatori/aplicații, în loc să consultăm repetat tabelele persistente, le putem „menaja" creând una sau mai multe tabele temporare pe care să le stoarcem (pe îndelete) de informații. Pe toată

durata „vieţii", liniile tabelelor temporare rămân neschimbate, deci rezultatele sunt corecte (consistente). În capitolul 14 vom vedea cum pot fi create în SQL tabelele temporare.

Tabele virtuale

În engleză, sintagma pentru acest tip de tabelă este *view*, adică *vedere* sau *imagine*. De aceea, în lucrările româneşti pot fi întâlnite mai multe titulaturi. O tabelă virtuală stabileşte o legătură semantică (printr-o expresie relaţională sau interogare) între tabele persistente şi/sau alte tabele virtuale, neavând tupluri proprii, ca o relaţie de bază (statică). Conţinutul (instanţierea) tabelei virtuale depinde, la un moment oarecare dat, de conţinutul tabelelor de bază din care derivă. Pentru a înţelege mai bine diferenţa, explicaţia se poate rezuma astfel: *tabela virtuală este cea pentru care pe disc se memorează numai schema, nu şi conţinutul.*

Ca şi tabelele temporare (şi cele „obişnuite"), definiţia unei tabele virtuale se păstrează pe disc. Înregistrările unei tabele temporare sunt stocate până la finalizarea modulului/blocului de cod (cazul celor locale) sau sesiunii (cazul celor globale), în timp ce înregistrările unei tabele virtuale nu se păstrează. Şi tabelele temporare, şi cele virtuale pot fi actualizate. Actualizarea nu ridică nicio problemă în cazul tabelelor temporare. În schimb, actualizarea unei tabele virtuale trebuie să se propage în tabelele persistente din care provind tuplurile acesteia. Sunt situaţii destul de frecvente în care o modificare a unei linii într-o tabelă virtuală nu poate fi „tradusă" cu exactitate în modificări ale tabelelor de bază, cazuri în care, de obicei, aceste modificări sunt respinse de către SGBD.

Tabelele virtuale oferă oricărui utilizator al unei baze de date posibilitatea prezentării datelor în funcţie de nevoile sale specifice. De asemenea, raţiuni de securitate şi confidenţialitate a anumitor informaţii pot conduce la izolarea unor date faţă de utilizatorii neautorizaţi, lucru deplin posibil prin intermediul *view*-urilor. Pornind de la aceleaşi tabele de bază, se pot crea oricâte tabele virtuale, în funcţie de situaţie. Tabelele virtuale constituie suportul creării schemelor externe. Odată definită, o tabelă virtuală poate fi referită ca o tabelă de bază oarecare.

Proceduri stocate

O procedură stocată este o secvenţă de program (cod) a cărei definiţie/semnătură face parte integrantă din baza de date. Avantajele utilizării procedurilor stocate decurg din faptul că acestea sunt parte din structura (schema) bazei, fiind păstrate în dicţionarul de date (catalogul sistem). Există mai multe tipuri de proceduri stocate: funcţii pentru calculul unor valori implicite, proceduri/funcţii de validare la nivel de linie sau înregistrare, funcţii/proceduri de calcul a unor expresii complexe etc. Începând cu versiunea 1999 a standardului, SQL-ul prezintă o componentă importantă - PSM (Persistent Stored Modules), care încearcă să instituie un etalon în redactarea procedurilor stocate. Unificarea PSM este o sarcină delicată. SQL-ul a apărut şi s-a consacrat ca limbaj *neprocedural*, care nu lucrează cu module de program clasice. Aşa că fiecare producător de SGBD-uri şi-a perfecţionat

unul (sau chiar mai multe) limbaje procedurale. Alinierea la specificațiile PSM este un proces de durată și oarecum de uzură (dar nu pentru noi).

Declanşatoare

Declanşatorul (*trigger*) a apărut ca un tip special de procedură stocată care este executată automat când un eveniment predefinit (inserare, actualizare sau ştergere) modifică o tabelă. Utilitatea declanşatoarelor este evidentă la formularea unor restricții mai complexe decât „suportă" comenzile CREATE/ALTER TABLE, cum ar fi: actualizarea automată a unor atribute calculate, jurnalizarea modificărilor suferite de baza de date, păstrarea integrității referențiale etc.

Există diferențe sensibile între SGBD-uri şi în ceea ce priveşte sintaxa declanşatoarelor, deoarece acestea sunt redactate în extensiile procedurale ale SQL, care prezintă diferențieri majore de la un producător la altul. În ultimii zece ani, au apărut şi alte evenimente-declanşatoare în afara celor trei, cum ar fi: conectarea la baza de date care echivalează cu deschiderea unei sesiuni de lucru, închiderea sesiunii, crearea unui obiect în BD, pornirea sau oprirea bazei de date etc.

Secvențe

Secvența este un alt tip de obiect al unei BDR care nu face parte, propriu-zis, din modelul relațional. A apărut din necesitatea asigurării unicității cheilor primare/alternative. Să luăm cazul atributului CodCl din tabela CLIENȚI. La inserarea unui client, valoarea sa pentru atributul CodCl ar trebui să fie superioară celei a precedentului client introdus sau, în orice caz, dacă nu superioară, măcar diferită. Dacă ar fi un singur utilizator cu drept de introducere a firmelor-client în BD, lucrurile ar fi relativ simple, corectitudinea valorilor CodCl depinzând de atenția sa. Acest scenariu este cu totul improbabil în aplicațiile pentru afaceri, în care pot exista zeci, chiar sute sau mii de utilizatori ce lucrează simultan cu baza, iar, mai ales, în primele săptămâni de la darea în folosință a aplicației, clienții pot fi introduşi de către mai mulți utilizatori (uneori chiar simultan).

Secvența este un obiect care, la fiecare invocare a sa, generează o valoare unică, indiferent de câte cereri/invocări simultane sunt. Crearea unei secvențe presupune, în afara numelui său, precizarea minimului, maximului şi a incrementului valorilor generate.

2.4. SGBDR-uri şi servere de baze de date relaționale

Până acum am folosit cei doi termeni, *SGBD* şi *server de baze de date*, fără a avea prea mare grijă în a preciza ce îi aseamănă şi ce îi diferențiază. Nici de acum încolo nu vom fi prea grijulii, stabilind însă, din capul locului, că orice server de bază de date (server BD) este un SGBD, însă nu orice SGBD este server BD.

2.4.1. Scurtă istorie „veche" a SGBDR-urilor

În prima ediţie a lucrării am evocat principalele repere cronologice în realizarea celor mai importante SGBD-uri relaţionale. Acum o să scurtăm serios povestea. Lucrările fondatoare ale lui E.F. Codd au fost primite cu entuziam reţinut de către firma la care lucra acesta - IBM. Rezerva IBM era legată nu numai de faptul că era vorba despre bazele unei tehnologii noi, nici cei mai buni prezicători neavând cum să ghicească dacă relaţionalul se va dovedi viabil sau va fi un eşec. IBM-ul avea, la acel moment, unul dintre cele mai puternice SGBD-uri de pe piaţă - IMS (ce gestiona baze de date ierarhice), în care investise mulţi bani şi oameni. Or, afirmaţiile lui Codd la adresa IMS-ului, făcute deseori în prezenţa clienţilor, nu erau dintre cele mai măgulitoare, ceea ce a iritat o serie de „feţe bisericeşti" ale companiei[14].

După câţiva ani, IBM se hotărăşte să aloce bani într-un proiect care să se materializeze într-un SGBD relaţional - System/R. Proiectul s-a derulat în laboratorul Santa Teresa din San José, California. Prima fază - anii 1974 şi 1975 - s-a concretizat într-un prototip al SGBDR-ului, care a fost rescris în anii 1976 şi 1977 pentru a permite interogarea simultană a mai multor tabele şi accesul multi-utilizator. IBM a distribuit produsul pe parcursul anilor 1978 şi 1979 la câţiva colaboratori ai săi pentru evaluări, dar în 1979 firma abandonează proiectul System/R, considerându-l nefezabil[15]. System/R a fost însă punctul de plecare pentru realizarea unuia dintre cele mai cunoscute SGBDR-uri, DB2.

Între timp, au fost realizate alte SGBD-uri, fie în stadiu de prototip, fie finisate de-a binelea: INGRES (Interactive Graphics and Retrieval System), al Universităţii Berkeley, QUERY BY EXAMPLE, Centrul de Cercetări T.J. Watson, PRTV (Peterlee Relational Test Vehicle), Centrul Ştiinţific Peterlee (Marea Britanie), patronat de firma IBM, RDMS (Relational Data Management System), General Motors, INFORMIX (denumit iniţial Relational Data Systems) etc.

Ca reper, reţinem şi anul 1977, când un grup de ingineri de la Menlo Park, California, a fondat compania Relational Software Inc., cu scopul declarat de a dezvolta un SGBD bazat pe limbajul SQL. Produsul, denumit Oracle, a fost lansat în 1979 şi a reprezentat primul SGBDR comercial. Dincolo de faptul că Oracle precedă cu doi ani primul SGBDR comercial al firmei IBM, orientarea sa a fost extrem de bine gândită, rulând pe minicalculatoarele VAX (Digital Equipment Corporation), sensibil mai ieftine decât mainframe-urile IBM, şi deci cu un segment de piaţă mai mare. Astfel, succesul a fost considerabil. Astăzi, firma, redenumită Oracle Corporation, este liderul produselor software dedicate mediilor de lucru cu bazele de date.

Primul SGBDR comercializat de IBM a fost SQL/DS (DS - Data System) anunţat în 1981 şi comercializat în 1982. În 1983, IBM a lansat DB2, dedicat iniţial

14. Evocarea are o uşoară nuanţă de poveste. Eu am auzit-o de la Fabian Pascal, care o preluase de la Chris Date.

15. Pentru amănunte privind odiseea System R, vezi şi documentele de la adresa web: http://www.mcjones.org/System_R/SQL_Reunion_95

mainframe-urilor sale, dar care în timp a fost portat pe toate sistemele de operare importante.

Până la jumătatea anilor '80, SGBD-urile relaţionale au stat în umbra celor ierarhice şi reţea. Motivul principal l-a constituit viteza mai mică, parametru esenţial în sistemele „confruntate" cu accesul simultan a sute sau chiar mii de utilizatori. În a doua parte a anilor '80, performanţele SGBDR-urilor au fost net ameliorate. Creşterea vitezei, care s-a adăugat atuului principal, interogarea ad-hoc prin utilizarea limbajelor de generaţia a IV-a (SQL, QBE, Quel), a lărgit continuu segmentul ocupat de SGBDR-uri.

Odată cu apariţia PC-urilor, tehnologia relaţională a devenit accesibilă unei largi categorii de utilizatori. Din anii '80 încoace, produse precum dBase (Ashton-Tate, Borland), Rbase (Microrim), Clipper (Nantucket, Computer Associates), DBFast (Computer Associates), Foxbase-FoxPro (Fox Software, Microsoft), Paradox (Borland, Corel), Approach (Lotus, IBM), Access (Microsoft) au fost folosite de milioane de deţinători de calculatoare personale, de la ignoranţi, până la profesionişti în dezvoltarea de aplicaţii cu baze de date.

Este drept, deşi categoria începătorilor în ale bazelor de date ne interesează - nu numai ca posibili cumpărători ai cărţii, cât mai ales ca persoane dornice de însuşirea unor cunoştinţe pentru a-şi folosi optim bazele de date proprii (şi mai ales ca să-şi dorească mai mult de la acest domeniu) -, în capitolele care urmează nu vom zăbovi asupra dialectelor SQL ale acestor SGBD-uri, ci ale patru dintre cele mai importante servere BD actuale.

2.4.2. SGBD-uri non-servere BD

Cele mai simple şi ieftine SGBDR-uri sunt cele pentru uz individual sau aplicaţii de tip file-server. Exemplele clasice sunt Visual FoxPro, Paradox şi Access. Visual FoxPro-ul a avut în ţara noastră un mare succes, comparabil doar cu cel din câteva ţări est-europene. Modul în care calculatoarele şi reţelele de calculatoare au pătruns şi s-au extins la noi explică adoptarea sa în multe aplicaţii de întreprindere şi predarea sa în şcoli, licee şi universităţi. Generaţia mea de truditori în ale bazelor de date este, în bună parte, generaţia Foxpro (vezi [Fotache *et al.*, 2002]). Astăzi, termenul este frecvent folosit în sens peiorativ, dar la timpul său (la începutul anilor '90), cu greu se putea găsi la noi ceva mai bun sau mai ieftin (şi uşor de copiat).

În prezent, Visual FoxPro (vezi figura 2.11), ajuns la versiunea 9, a ieşit la pensie. Microsoft-ul s-a hotărât să oprească producţia şi suportul pentru acest SGBD venerabil. Semnele rele erau vizibile de mai bine de zece ani. Microsoft nu a mai venit cu modificări importante legate de gestiunea bazelor de date în FoxPro de la versiunea Visual FoxPro 5, lansată în 1994. În versiunile ulterioare s-au mai corectat câteva erori, s-au mai introdus câteva opţiuni privitoare la nucleul SQL (mai ales la versiunea 9) şi... cam atât. Este adevărat, VFP nu este doar un SGBD, ci un mediu integrat de dezvoltare de aplicaţii, cu foarte bune generatoare de rapoarte, meniuri şi formulare şi un limbaj de programare care a fost orientat pe obiecte cu mult înaintea Visual Basic-ului.

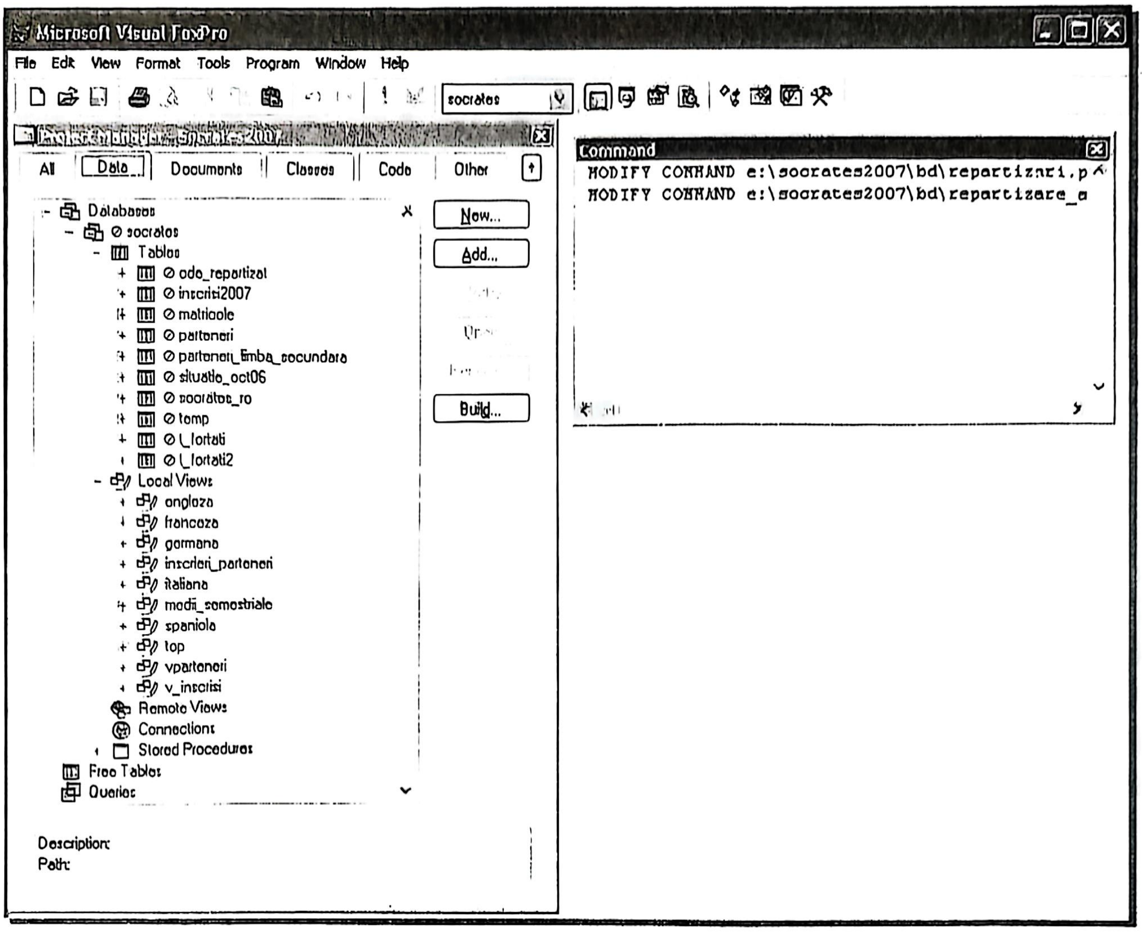


Figura 2.11. Afişarea conţinutului unui proiect (inclusiv schema BD) în Visual FoxPro

Totuşi, pentru a intra în lumea bună a serverelor BD, VFP trebuia înzestrat cu opţiuni de creare a utilizatorilor[16], a profilurilor, de alocare şi revocare de drepturi pe toate obiectele bazei de date. Mai mult, ar fi trebuit să permită „nativ" arhitecturi client/server şi web[17], module de arhivare, restaurare şi replicare, dar mai ales să se descurce cu recuperarea bazei în cazuri de dezastre (întreruperi de curent, blocări ale Windows-ului). Or, responsabilii de aplicaţii complexe realizate în VFP, cu zeci de utilizatori conectaţi simultan la BD, ştiu ce este adrenalina când „pică nişte indexuri"[18].

---

16. Până şi Access-ul are, de câţiva ani, opţiuni de creare de utilizatori şi chiar de replicare.
17. Există opţiuni de adaptare a VFP la arhitecturi apropiate de cea c/s şi web (vezi [Fotache *et al.* 2002], însă efortul dezvoltatorilor de aplicaţii este considerabil.
18. Vorbesc din propria experienţă. Cea mai mare jenă profesională (de până acum) mă urmăreşte de vreo zece ani, când, împreună cu alţi colegi de la Catedra de Informatică Economică, am încercat să modific modul de înscriere la concursul de admitere al FEAA Iaşi. Ghinionul a fost că decanul a avut încredere în noi, dar, contrar testelor preliminare, în primele patru ore de la începerea înscrierilor (se adunaseră vrei două sute de persoane) ne-au picat indexurile de vreo zece ori. Ulterior, am aflat că dintre cele 20 de staţii (PC-uri) de înscriere, două erau „mufate" un pic greşit (şi, absolut aleatoriu, pierdeau din când în când legătura cu serverul pentru câteva fracţiuni de secundă).

Deşi i-a fost anunţat sfârşitul, mai am o speranţă pentru VFP, şi anume ca, odată ce Microsoft îşi va lua mâinile de pe produs, să intre în lumea open source şi să cunoască o nouă viaţă (ar fi, cred, a treia...).

Cel mai celebru SGBD din categoria non-serverelor BD este Access, copilul pe care Microsoft-ul l-a preferat în dauna Visual FoxPro-ului[19]. Access-ul are toate atuurile care l-au făcut aşa de popular, de la uşurinţa folosirii până la integrarea cu celelalte aplicaţii din Office şi migrarea naturală la SQL Server. Şi Access este mai mult decât un SGBD – vezi figura 2.12.

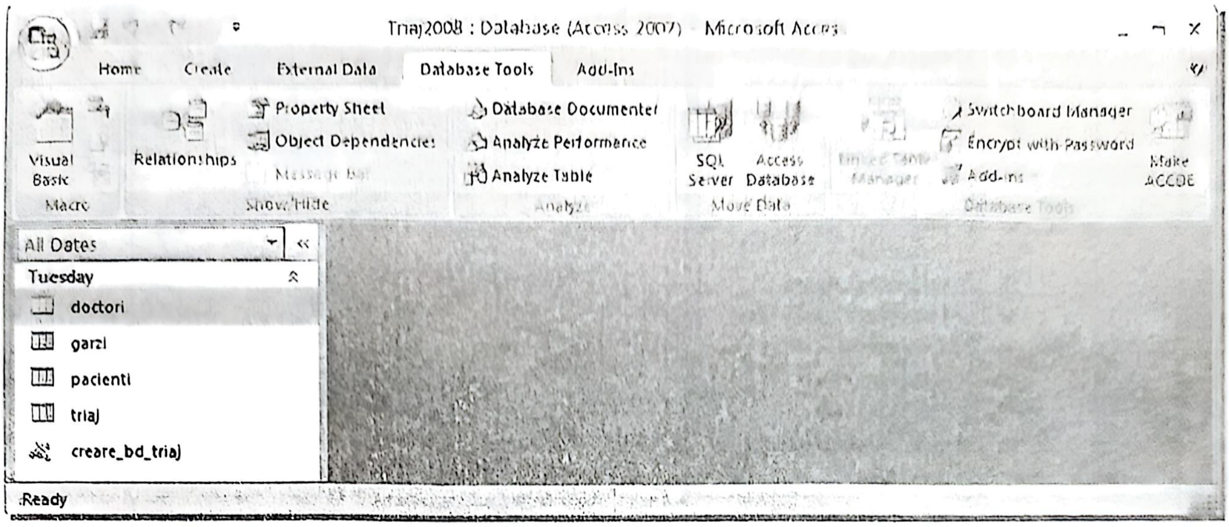


Figura 2.12. Afişarea conţinutului unui proiect (inclusiv tabelele BD) în Access

Însă, deşi i-au fost adăugate funcţiuni redutabile, Access nu este (încă) un server BD, iar dialectul său SQL este unul rezonabil, dar nu impresionant, în niciun caz superior celui din VFP.

2.4.3. Servere de baze de date

Aplicaţiile complexe din bănci, corporaţii, organizaţii şi instituţii gestionează volume uriaşe de informaţii şi sunt operaţionale prin efortul a mii de utilizatori ce lucrează simultan cu baza de date. Cerinţele de viteză şi fiabilitate sunt maxime. Practic, multe aplicaţii nu au niciun moment de pauză, orice întrerupere fiind foarte gravă. Vorbim despre lumea *aplicaţiilor critice*, cu regulile ei foarte dure, iar SGBD-urile care constituie parte din infrastructura acestor aplicaţii sunt cele de categorie grea: Oracle (Oracle), DB2 (IBM), Informix (IBM), Sybase (Sybase), SQL Server (Microsoft) etc. Fiind mult mai robuste, fiabile, sunt şi mult mai costisitoare.

19. În fine, poate că exprimarea e cam melodramatică, dar ca fox-ist (şi ofticos pe deasupra) n-am putut să mă abţin.

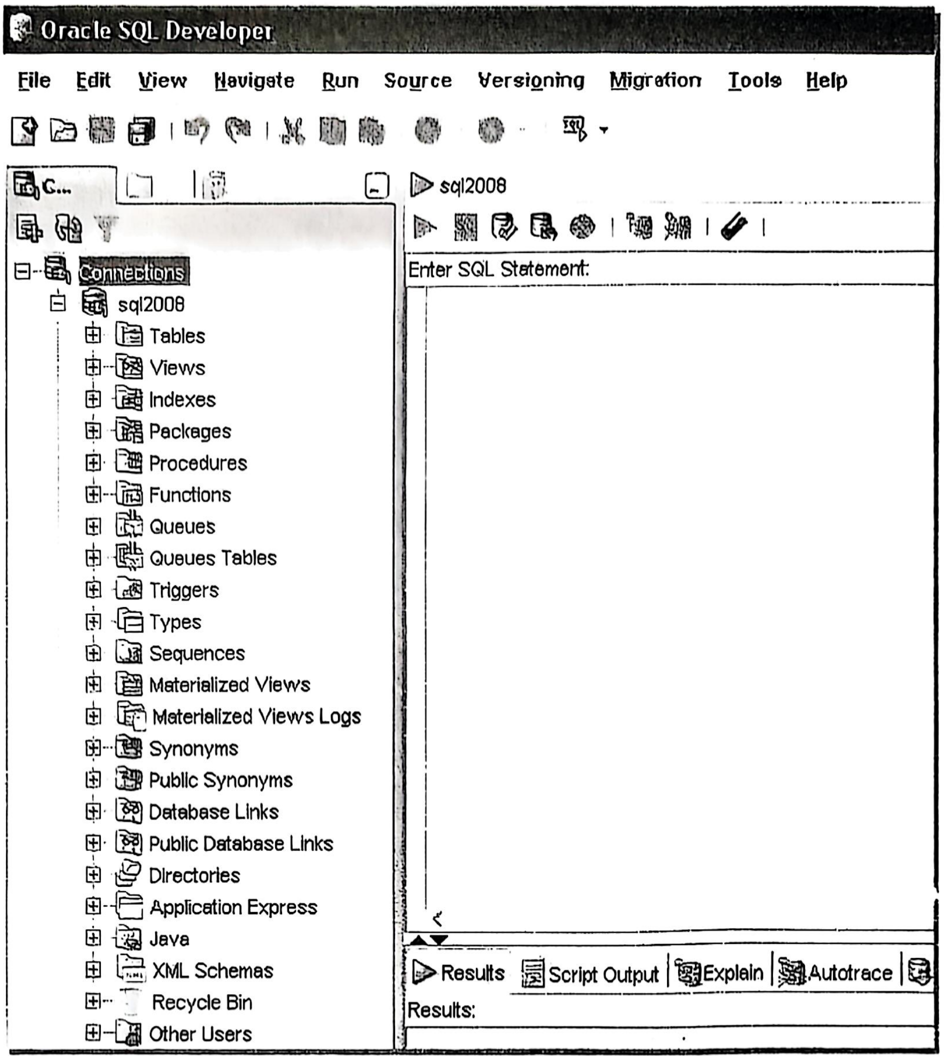


Figura 2.13. Obiecte dintr-o bază de date Oracle (afişate de „clientul" SQL Developer)

Titlul figurii 1.6 făcea trimitere la structura unui SGBD. De fapt, numai serverele BD sunt construite pe arhitectura respectivă. Conform standardului SQL [ISO/IEC 2007-1], un server SQL:

- gestionează sesiunile SQL care se derulează între server şi un client prin intermediul unei conexiuni SQL;
- execută comenzile SQL primite de la client, recepţionează şi transmite datele cerute;
- menţine starea sesiunilor SQL, inclusiv a identificatorului de autorizare şi a altor parametri impliciţi ai sesiunii.

Deşi au existat contradicţii între cifrele avansate de diferite firme de estimare a pieţei bazelor de date, astăzi majoritatea lucrărilor indică drept lider firma Oracle şi produsul său cu acelaşi nume. Figura 2.13 afişează imaginea schemei BD aşa cum apare în „clientul" (de-acum „clasic") SQL Developer. Serverul BD Oracle (cea mai recentă versiune a sa fiind Oracle 11g) este un etalon în măsurarea puterii şi funcţionalităţii unui SGBD, fiind folosit în mai toate tipurile de organizaţii, de la aplicaţii pentru firme de mărime medie până la aplicaţii pentru corporaţii multinaţionale. Tradiţional un produs scump, sub presiunea concurenţilor şi

serverelor BD open source, Oracle dispune astăzi de o politică de licenţiere mult mai flexibilă.

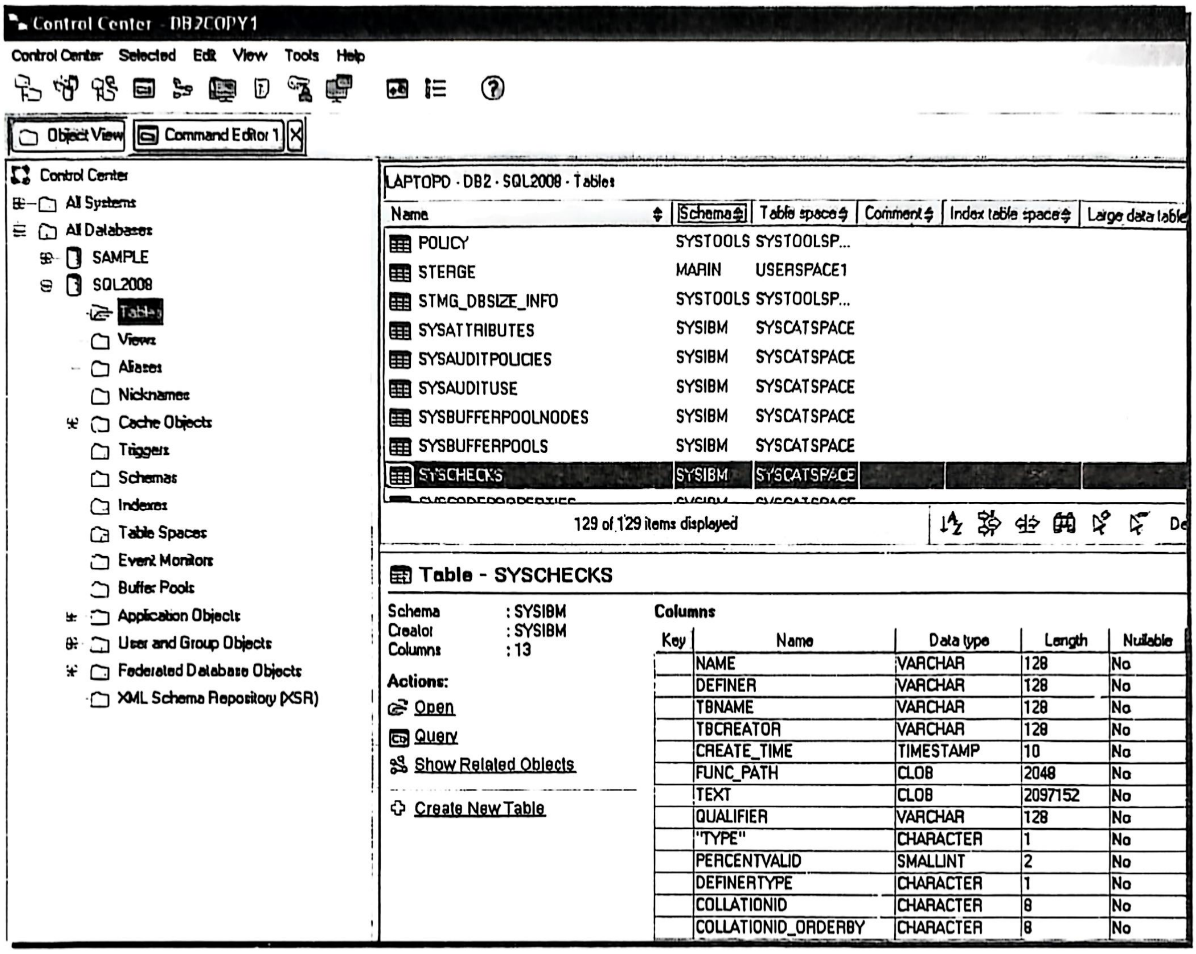


Figura 2.14. Obiecte dintr-o bază de date DB2

Cel mai important concurent al Oracle pe piaţa serverelor BD este produsul DB2 al firmei IBM, ajuns la versiunea 9.5. Deşi are o imagine foarte bună şi un număr impresionant de instalări în SUA şi alte ţări dezvoltate, în România prezenţa DB2 este mai degrabă discretă. Figura 2.14 ilustrează modul în care sunt afişate în DB2 (modulul Control Center) obiectele unei baze de date.

Pentru noi, DB2 prezintă o importanţă grozavă nu atât datorită pieţei acoperite, cât rolului imens pe care l-a jucat firma IBM în procesul de standardizare a SQL-ului, cel puţin primele două standarde majore, SQL-89 şi SQL-92, fiind, în mare măsură, copii ale dialectului SQL din DB2, aşa cum se prezenta la acea dată.

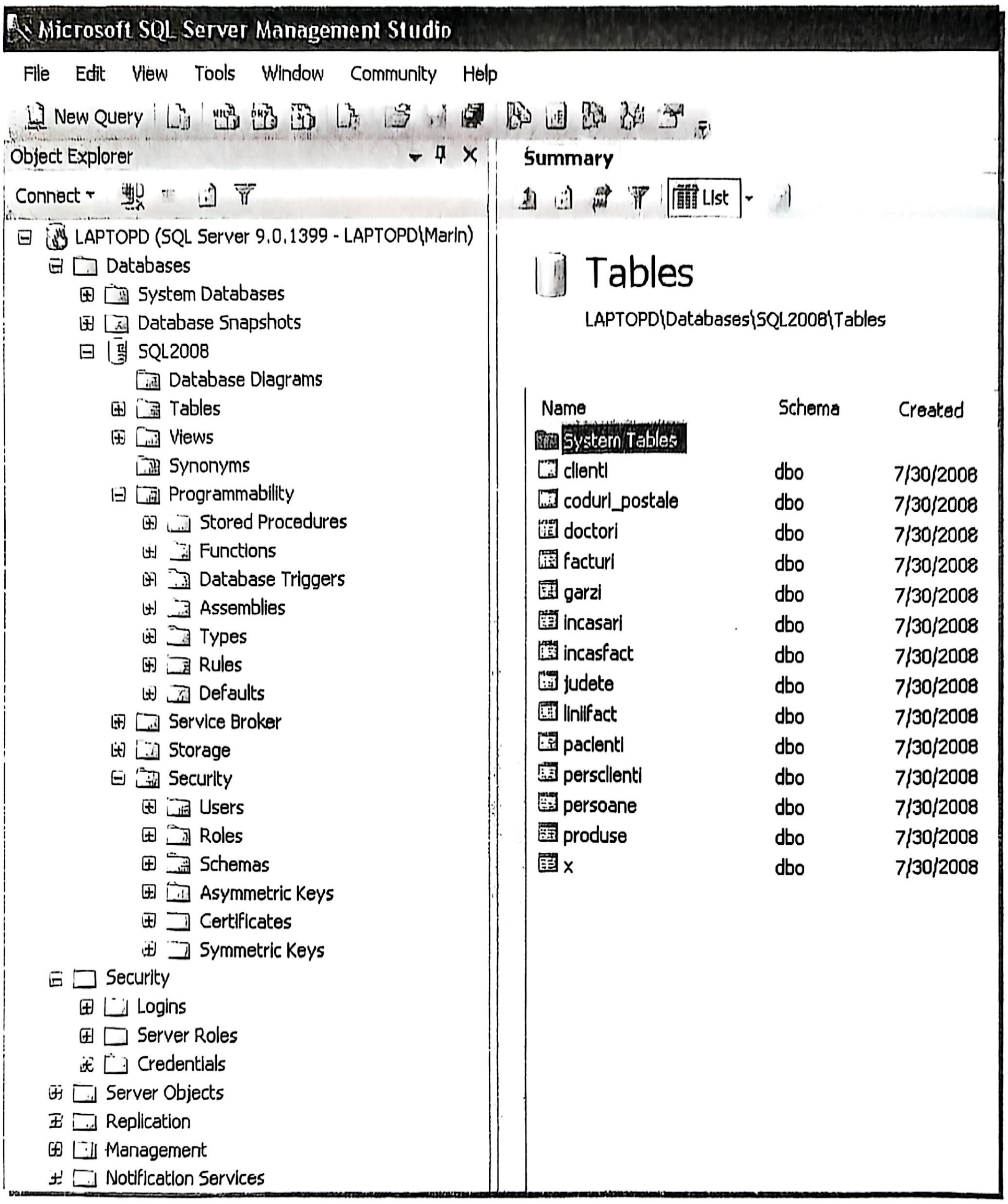


Figura 2.15. Obiecte dintr-o bază de date MS SQL Server

În ultimul deceniu, dintre serverele „proprietare", cea mai dinamică evoluţie nu a avut-o nici DB2, nici Oracle, ci produsul Microsoft denumit chiar SQL Server - vezi figura 2.15. Titulara produsului este un pic nepoliticoasă (probabil că cea mai decentă denumire ar fi fost *One of the SQL Servers*), dar pe piaţă a prins[20].

Dinamica a fost asigurată şi de faptul că produsul a apărut spre mijlocul anilor '90 (baza fiind deci foarte mică). Spre deosebire de giganţii DB2/Oracle, MS SQL Server a pornit-o „de jos", ca server BD pentru grupuri de lucru, ideal pentru aplicaţii folosite de grupuri mici şi mijlocii de utilizatori. Preţul a fost, de la început, extrem de tentant şi continuă să fie cu mult sub cele ale DB2 şi Oracle. Cu fiecare

20. SQL Server a fost dezvoltat, iniţial, printr-un parteneriat cu firma Sybase.

versiune, ultima fiind cea din 2008[21], SQL Server a continuat să câştige în funcționalitate şi fiabilitate, astfel încât astăzi poate fi folosit şi la aplicații de mari dimensiuni.

În materie de opțiuni SQL, produsul Microsoft a prezentat destul de multe diferențe față de standard. Pe parcursul capitolelor următoare o să observăm multe funcții şi opțiuni „proprietare". Tendința de aliniere la standardele recente (SQL:1999, SQL:2003, SQL:2008) ale SQL este vizibilă, însă cu fiecare nouă versiune poate fi una dintre explicațiile succesului comercial al SQL Server-ului.

Ultimii 15 ani au consacrat un segment de piață care se constituie ca o alternativă atrăgătoare la primadonele serverelor BD - serverele BD open source. Inspirate din succesul sistemelor de operare de tip open source şi, ulterior, serverelor web, serverele BD din această categorie au atras atenția multor dezvoltatori de aplicații, mai ales în condițiile ubicuității arhitecturilor pe trei sau mai multe straturi.

Este greu de spus care este cel mai bun produs din această categorie. Dacă ar fi după numărul de utilizatori, probabil că cel mai bine plasat este MySQL[22]. Concurenții sunt, însă, foarte numeroşi şi puternici. Amintim doar PostgreSQL[23], Mini SQL (sau mSQL)[24], Ingres[25], MaxDB[26], Firebird[27]. Paradoxul este că MySQL-ul a devenit liderul pieții serverelor BD open source după 1995 în condițiile în care avea implementat unul dintre cele mai slabe dialecte SQL, iar procedurile stocate erau evocate doar din când în când pe grupurile de discuții. De la versiunea 5, MySQL-ul este mai apropiat de un nivel decent de opțiuni specifice serverelor BD.

Nivelul jenant (în materie de SQL) al MySQL-ului a fost unul dintre motivele alegerii (cu câțiva ani în urmă) a PostgreSQL-ului ca server BD open source reprezentativ pentru lucrul cu studenții. Deşi neocolit de probleme tehnice în a doua parte a anilor '90, astăzi PostgreSQL-ul, ajuns la versiunea 8.3, poate fi folosit în aplicații sofisticate, având un nivel de funcționalitate redutabil. Mai mult, în materie de aliniere la standardele SQL, PostgreSQL merită toate laudele.

Figura 2.16 ilustrează modul în care sunt prezentate obiectele unei baze de date în, probabil, cel mai popular „client" PostgreSQL - şi anume *pgAdmin* - client pe care îl vom folosi şi noi pentru exemplificarea comenzilor SQL şi a modulelor de proceduri stocate.

21. Comenzile SQL prezentate în carte au fost executate pe SQL 2005, deci produsul Microsoft a fost oarecum discriminat în privința ultimei versiuni.
22. www.mysql.com
23. www.postgresql.org
24. www.hughes.com.au
25. www.ingres.com
26. www.sdn.sap.com/irj/sdn/maxdb
27. www.firebirdsql.org

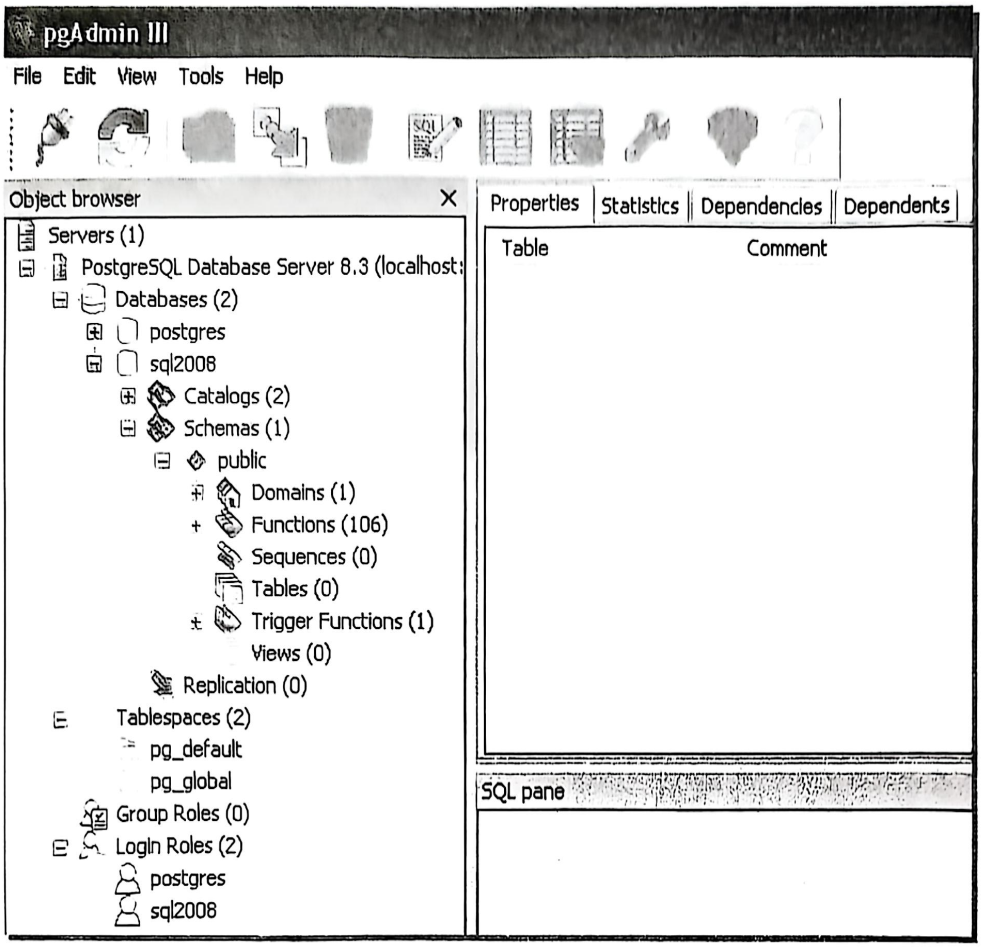


Figura 2.16. Obiecte dintr-o bază de date PostgreSQL afişate în clientul pgAdmin

Capitolul 3
Standarde şi dialecte SQL. Tipuri de date. Crearea tabelelor şi modificarea conţinutului

De obicei, noţiunea de *limbaj de programare* duce cu gândul la Pascal, C, C++, Basic, Fortran, COBOL, PL/1, Algol, Java etc. Iniţiaţii ştiu că mai există limbaje de asamblare şi limbaje-maşină. Toate aceste limbaje sunt generale, în sensul că permit realizarea de module ale oricărui strat dintr-o aplicaţie file-server, client/server sau web (vezi paragraful 1.2). Este foarte posibil ca meniurile unei aplicaţii să fie realizate în Visual Basic, o parte dintre formulare în C++, iar o altă parte în Visual Basic.

Cu bazele de date, am văzut că lucrurile au stat, încă de la începuturile acestora, un pic altfel. Niciunul dintre limbajele de uz general nu stă grozav în materie de comenzi DDL (Data Definition Language), DML (Data Manipulation Language) sau DCL (Data Control Language).

Astfel, toate SGBD-urile, servere BD sau nu, au dispus (şi dispun în continuare) de un limbaj propriu „înţesat" de comenzi DDL, DML şi DCL. Profunda incompatibilitate între limbajele numeroaselor SGBD-uri a atras preocupări majore pentru elaborarea unor standarde care să armonizeze „dialectele" acestora.

Dintre limbajele special dedicate bazelor de date, propuse din anii '60 încoace, unul a reuşit să se impună în toate serverele importante - limbajul SQL (al cărui nume este acronimul de la Structured Query Language). Succesul său fără echivalent în lumea bazelor de date l-a făcut pe Michael Stonebraker să califice SQL-ul, într-un acces de entuziam, drept limbaj intergalactic pentru date (*intergalactical dataspeak*) [Stonebraker *et al.* 1998].

După Groff şi Weinberg, principalele atuuri ale SQL sunt [Groff & Weinberg 2002]:

- independenţa de producător, nefiind o tehnologie proprietară. Prin urmare, modificarea, într-o aplicaţie, a serverului BD, nu este atât de traumatizantă;
- portabilitate între diferite sisteme de operare: astăzi SQL este implementat pe servere BD ce rulează pe PC-uri, mainframe-uri etc.;
- este standardizat, având recunoaştere planetară (sună cam pompos, dar afirmaţia este reală!);
- are sprijinul nu numai organismelor de standardizare, ci şi al marilor firme producătoare de instrumente pentru gestionarea BD: IBM, Oracle, Microsoft;
- „filosofia" sa se bazează pe modelul relaţional de organizare a datelor. Pentru unii, acest lucru ar putea fi un dezavantaj, însă structura tabelară a relaţiilor rămâne uşor de înţeles şi de deprins de către utilizatori;

- este un limbaj de nivel înalt, cu sintaxă apropiată de limba engleză, de aici relativa facilitate în deprinderea comenzilor;
- permite formularea de răspunsuri la numeroase interogări simple, ad-hoc, neprevăzute iniţial. Acesta rămâne încă un atu important al BD relaţionale în competiţia cu cele pur obiectuale;
- constituie suportul „programatic" pentru accesul la BD. Aproape orice modul al unei aplicaţii cu BD conţine comenzi SQL incluse în programe redactate în cele mai diverse limbaje, de la PHP la Java şi C++;
- permite imagini multiple asupra datelor din bază, fără a afecta cu nimic corectitudinea şi integritarea BD;
- este un limbaj relaţional complet. Pentru cei doi autori, noţiunea de completitudine se referă la faptul că, deşi a pornit de la interogarea BD, SQL se foloseşte pentru o largă categorie de operaţiuni: crearea de obiecte din bază, actualizarea datelor, crearea conturilor pentru utilizatori, acordarea de drepturi etc.;
- permite definirea dinamică a datelor, în sensul modificării structurii bazei chiar în timp ce o parte din utilizatori sunt conectaţi la BD;
- constituie un excelent suport pentru implementarea arhitecturilor client-server şi web.

Nu există astăzi lucrări sau publicaţii în domeniul SGBD-urilor care să nu prezinte mecanismul de lucru al acestui limbaj sau ultimele produse/tendinţe din lumea SQL. Impactul său este profund. SQL este, pe de o parte, unul dintre „responsabilii" noii apropieri a non-informaticianului de datele sale, iar, pe de altă parte, pentru profesionişti, reprezintă nucleul dezvoltării aplicaţiilor ce utilizează bazele de date.

Deşi este referit, în primul rând, ca un limbaj de interogare, SQL este mult mai mult decât un instrument de consultare a bazelor de date, deoarece permite, în egală măsură:

- definirea datelor;
- consultarea BD;
- manipularea datelor din bază;
- controlul accesului;
- partajarea bazei între mai mulţi utilizatori ai acesteia;
- menţinerea integrităţii BD.

Setul comenzilor de bază care îndeplinesc aceste funcţionalităţi este cuprins în tabelul 3.1.

Tabelul 3.1. Cele mai importante comenzi SQL

Comandă	**Scop**
Pentru manipularea datelor	
SELECT	Extragerea datelor din bază
INSERT	Adăugarea de noi linii într-o tabelă
DELETE	Ştegerea de linii dintr-o tabelă
UPDATE	Modificarea valorilor unor atribute

Pentru definirea bazei de date	
CREATE TABLE	Adăugarea unei noi tabele în baza de date
DROP TABLE	Ştergerea unei tabele din bază
ALTER TABLE	Modificarea structurii unei tabele
CREATE VIEW	Crearea unei tabele virtuale
DROP VIEW	Ştergerea unei tabele virtuale
Pentru controlul accesului la BD	
CREATE USER	Crearea contului unui utilizator
ALTER USER	Modificarea proprietăţilor contului unui utilizator
DROP USER	Ştergerea contului unui utilizator
GRANT	Acordarea unor drepturi la obiecte din bază pentru un cont de utilizator sau grupuri (roluri) de utilizatori
REVOKE	Revocarea unor drepturi
Pentru controlul tranzacţiilor	
COMMIT	Marchează sfârşitul „corect" al unei tranzacţii
ROLLBACK	Abandonează tranzacţia în curs

3.1. Standardizarea SQL

În privinţa standardizării SQL, lucrurile sunt destul de încâlcite. Putem vorbi despre standarde SQL, adică SQL-89, SQL-92, SQL:1999, SQL:2003, dar, la fel de bine, dacă ţinem cont că numărul alocat de ISO este acelaşi - 9075 -, putem vorbi, cel puţin la fel de îndreptăţiţi, despre versiunile SQL-89,... ale standardului SQL. Pentru a menţine deruta, noi vom folosi ambele exprimări.

O altă sursă de zăpăceală este că există în istoria SQL versiuni intermediare sau revizuiri minore ale standardului, aşa încât la un inventar ceva mai atent am putea completa pomelnicul: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008. Mutiplicarea (sub)versiunilor standardului are şi un efect benefic: nu ne mai simţim obligaţi să le ţinem minte pe toate. La drept vorbind, senzaţia de degringoladă vizavi de greu numărabilele versiuni este cauzată de faptul că standardul SQL nu mai este, începând cu varianta 1999, monolitic, ci compus dintr-o serie de părţi pe care le vom evoca peste câteva pagini. O variantă a standardului poate să însemne, de fapt, apariţia sau modificarea substanţială a uneia-două părţi, restul rămânând aproape intact.

3.1.1. Apariţia SQL-ului

La începutul anilor '70, piaţa era dominată de SGBD-urile care funcţionau după principiile modelelor ierarhice şi reţea. Liderul autoritar (concurenţii erau nesemnificativi în acea perioadă) al bazelor de date era firma IBM, care domina de

departe atât piaţa de hardware (celebra serie System/360 de mainframe-uri), cât şi pe cea de software.

În cele două articole ale inventatorului modelului relaţional [Codd 1969] [Codd 1970], acesta sugera „adoptarea unui model relaţional pentru organizarea datelor [...] care să permită punerea la punct a unui sublimbaj universal pentru gestiunea acestora, sublimbaj care să fie, în fapt, o formă aplicată de calcul asupra predicatelor". După cum vom vedea în capitolul următor, există două categorii majore de limbaje de interogare, una bazată pe algebra relaţională şi alta bazată pe calculul relaţional. Codd a mizat pe a doua categorie (la urma urmei, modelul relaţional este bazat pe predicate), însă câştig de cauză pe piaţă au avut limbajele algebrice (SQL).

Sunt poate două dintre motivele care explică ciudăţenia ca IBM-ul să nu includă în echipa de dezvoltare a SQL-ului tocmai pe părintele modelului relaţional. În plus, poate că şi neincluderea lui Codd în proiectul SQL a accentuat înverşunarea acestuia şi a relaţioniştilor împotriva limbajului.

Un prim moment important în naşterea SQL îl constituie lansarea proiectului System/R de către firma IBM, eveniment ce a avut loc în 1974 (vezi [Astrahan *et al.* 1976]). Tot în 1974, Chamberlin şi Boyce au publicat un articol [Chamberlin & Boyce 1974], în care este prezentat un limbaj structurat de interogare, denumit SEQUEL (Structured English as QUery Language). SEQUEL a fost gândit de fapt ca o interfaţă (API - Application Program Interface) a aplicaţiilor cu SGBD-ul System/R ([Melton & Simon 2002], p. 24).

În 1975, Chamberlin, Boyce, King şi Hammer redactează o lucrare dedicată sublimbajului SQUARE [Boyce *et al.* 1975], asemănător SEQUEL-ului, dar care utilizează expresii matematice, şi nu cuvinte din limba engleză. Autorii celor două studii au demonstrat că limbajele SEQUEL şi SQUARE sunt complete din punct de vedere relaţional. În 1976, o echipă de autori condusă de Chamberlin elaborează o nouă lucrare [Chamberlin *et al.* 1976], în care se face referire la SEQUEL 2, acesta fiind declarat limbaj de interogare al SGBD-ului System/R al firmei IBM. Chamberlin schimbă denumirea SEQUEL în SQL - Structured Query Language (Limbaj Structurat de Interogare)[1] în 1980, dar şi astăzi mulţi specialişti pronunţă SQL ca pe predecesorul său. Don Chamberlin a lucrat la IBM până în 2003 (din 2003 este IBM Fellow), fiind implicat în dezvoltarea mai multor versiuni ale SGBD-ului DB2, iar în ultimii ani este coautor la documente şi standarde XML (XQUERY).

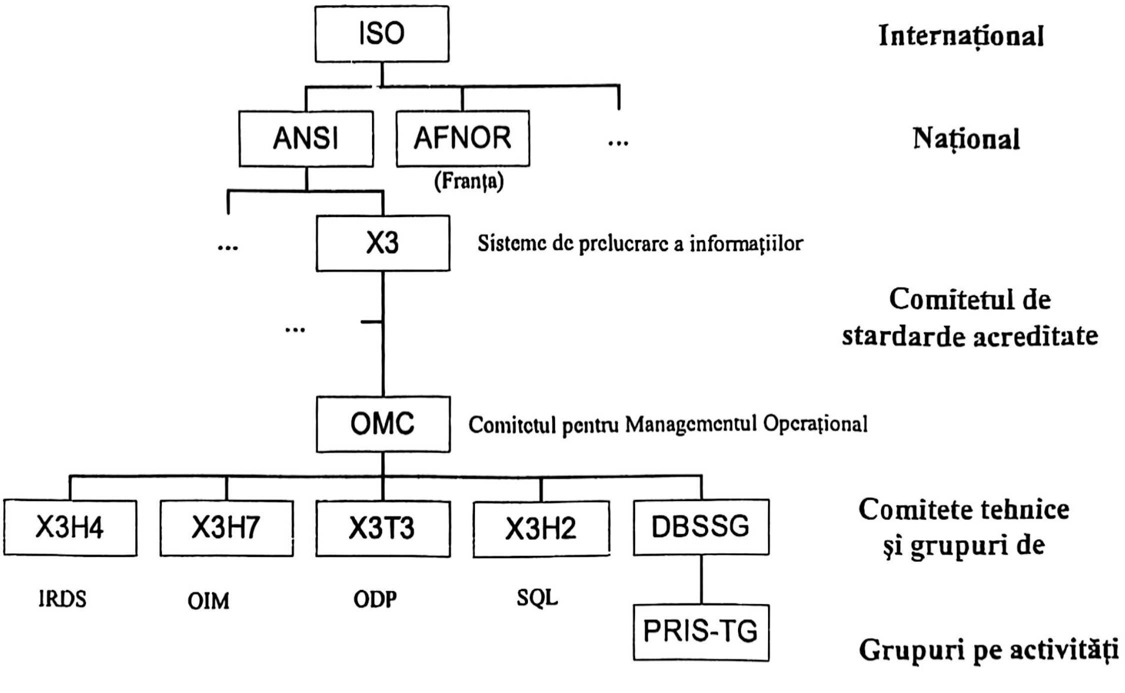
3.1.2. SQL-86 şi SQL-89, adică SQL1 şi SQL1.1

Stardardele internaţionale se redactează sub egida ISO (International Organization for Standardization). În materie de SQL, motorul standardizării a fost şi este ANSI (American National Standards Institute).

1. Se pare că a fost şi o problemă de copyright în privinţa sintagmei SEQUEL (vezi [Hernandez & Viescas 2000], p. 52).

Încercând să răspundă solicitărilor privind standardizarea unui limbaj de lucru cu bazele de date, ANSI a încredinţat această sarcină comitetului X3H2 în anul 1982. Comitetul pentru bază de date X3H2 (unul din numeroasele consilii tehnice ale X3) a fost (şi este, dar sub un alt nume, după cum o să vedem ceva mai jos) format din experţi independenţi şi din reprezentanţi ai firmelor importante din domeniul bazelor de date. Figura 3.1 ilustrează poziţia X3H2 în organismele de standardizare în anii '80 şi '90[2].

În această schemă, un standard nou începe cu o schiţă, *proiect de lucru* (working draft) întocmit de comitet (X3H2), proiect ameliorat până când membrii convin că este aproape de finalizare. Proiectul de lucru devine *proiect al comitetului* (committee draft) care este transmis comunităţii profesionale şi firmelor pentru comentarii şi propuneri. După preluarea observaţiilor şi propunerilor, proiectul comitetului este publicat sub titulatura *proiect internaţional de standard* şi, în final, *standard internaţional.*



Legendă:
ISO - International Organisation for Standardization
ANSI - American National Standard Institute
AFNOR - Association Française de Normalisation
OMC - Operational Management Committee
DBSSG - Database Systems Study Group
PRIS-TG - Predictable Real-time Information Management Task Group

Figura 3.1. Structura organismelor de standardizare
în domeniul bazelor de date în anii '80 şi începutul anilor '90

Comitetul X3H2 a primit responsabilitatea elaborării unui standard pentru un limbaj dedicat bazelor de date în anul 1982. Rapid (în circa doi ani), au fost elaborate câteva versiuni ale standardului, bazate din plin pe specificaţiile IBM

2. Adaptat după [Fortier 1999], p. 4.

SQL/DS şi DB2. Fiind considerat a fi mult prea impregnat de IBM, în 1984 standardul SQL a fost reproiectat pentru a i se asigura un minimum de generalitate.

ANSI publică în 1986 standardul *SQL ANSI X3.135-1986*, care utilizează, pentru reprezentarea sintaxei, forma Backus-Naur. Este un standard care continuă să se bazeze, într-o mare măsură, pe „dialectul" SQL al IBM şi se întinde pe aproximativ 100 de pagini. Organizaţia Internaţională pentru Standarde (ISO) a adoptat propriul document, aproape identic cu ANSI SQL-86, pe care l-a publicat în 1987 ca *ISO 9075-1987 Database Language SQL*.

SQL-86 defineşte comenzile de bază ale SQL, inclusiv pentru crearea de tabele şi tabele virtuale (CREATE TABLE, CREATE VIEW), însă nu conţine opţiuni de modificare a structurii sau ştergere (ALTER.../DROP...) şi nici comenzi pentru acordare şi revocare de drepturi utilizatorilor (GRANT/REVOKE)[3]. Lipsa facilităţilor privind definirea şi asigurarea restricţiilor referenţiale a generat discuţii aprinse şi critici vehemente, astfel încât, rapid, a fost publicat un set de specificaţii numit *Integrity Enhancement Feature* (elemente de ameliorare a integrităţii) prin care se pot defini chei primare şi chei străine ca elemente componente ale schemei bazei de date.

La trei ani de la publicarea SQL-86, prin revizuirea şi extinderea sa, se „naşte" SQL-89, care mai este denumit, în majoritatea lucrărilor de profil, SQL-1[4] sau SQL-89 - ANSI X3.135-1989, respectiv ISO 9075:1989. Deşi recunoscut ca fundament al multor SGBDR-uri comerciale, şi SQL1.1 şi-a atras numeroase critici. În plus, variantele comercializate de diferiţii producători, deşi esenţialmente asemănătoare, erau (şi sunt) incompatibile la nivel de detaliu. În afară de setul de specificaţii mai sus amintit, SQL-89 a inclus şi specificaţiile pentru apelul comenzilor şi funcţiilor SQL din limbaje-gazdă, precum COBOL, Fortran şi C.

Pe lângă ANSI, ale cărui standarde au cea mai largă audienţă, au fost înfiinţate şi alte organisme de standardizare SQL. X/Open este un grup de firme europene care a adoptat SQL ca nucleu al unei întregi serii de standarde menite să asigure realizarea unui mediu general pentru aplicaţii portabile, grefat pe sistemul de operare UNIX. IBM a avut un aport incontestabil la apariţia şi maturizarea SQL, fiind un producător cu mare influenţă în lumea SGBD-urilor, iar produsul său, DB2, a fost şi este un element de referinţă al SQL.

În 1989, un grup de producători de instrumente dedicate bazelor de date au format *SQL Access Group*, în vederea realizării conexiunilor dintre SGBDR-urile fiecăruia, pe baza unor specificaţii comune, din care un prim set a fost publicat în 1991 sub titulatura RDA (Remote Database Access). Specificaţiile RDA n-au reuşit să se impună pe piaţa SGBD-urilor. La insistenţele firmei Microsoft, SQL Access Group şi-a concentrat eforturile pentru elaborarea unei interfeţe-standard pentru SQL. Pe baza unui set de propuneri înaintat de companie, în 1992 au rezultat

3. Pentru detalii privind istoria SQL, vezi şi cuvântul înainte al lui Ken Jacobs la lucrarea [Hernandez & Viescas 2000], pp. 52-63.
4. Eu îmi asum „notaţia" din [Kriegel & Trukhnov 2008], oarecum „minoritară", prin care SQL-86 este referit ca SQL1, iar SQL-89 drept SQL1.1.

specificaţiile CLI (Call Level Interface). Având drept reper CLI, Microsoft a elaborat şi implementat în acelaşi an un set propriu, ODBC (Open DataBase Conectivity), care a devenit standard în materie de interfaţă SQL pentru accesarea diferitelor baze de date.

3.1.3. SQL-92

Pentru a umple golurile SQL1.1, ANSI şi ISO au elaborat în 1992 versiunea SQL-2, ANSI X3.135-1992 (*Database Language SQL*), respectiv ISO/IEC 9075:1992, specificaţiile fiind prezentate la un nivel mult mai detaliat (dacă SQL1 se întindea pe numai 100 de pagini, SQL-92 a fost publicat în aproape 600). Dintre numeroasele facilităţi aduse de SQL-92, merită amintite cu deosebire: joncţiunea externă (OUTER JOIN), atribute de tip dată-oră şi de alte tipuri, raportare standardizată a erorilor, un set standard de tabele din catalog (dicţionarul de date), modificarea schemei bazei (DROP, ALTER, GRANT, REVOKE), SQL dinamic, modificări şi ştergeri referenţiale în cascadă, amânarea verificării restricţiilor, nivele de consistenţă a tranzacţiilor etc.

Standardul SQL-92 defineşte trei nivele de conformitate:

- *Entry* – intrare, de bază (opţiunile din SQL-89 corectate, plus câteva mici adăugiri);
- *Intermediate* – intermediar, ce include aproximativ jumătate dintre facilităţi, cum ar fi: denumirea restricţiilor, suport pentru şiruri de caractere de lungime variabilă şi pentru seturi de caractere „naţionale", structuri de control de tip CASE, funcţii de conversie (CAST), operatorii INNER şi OUTER JOIN, SQL dinamic, operatorii ansamblişti (UNION, INTERSECT, EXCEPT) ([Kriegel & Trukhnov 2008], p. 24);
- *Full* – deplin, dedicate unor opţiuni avansate: restricţii amânabile, aserţiuni, tabele temporare locale, domenii etc. ([Kriegel & Trukhnov 2008], p. 24).

Fiecare producător îşi declară nivelul de conformitate al SGBD-ului în raport cu SQL-92. Spre exemplu, nucleul SQL din Oracle 8 era conform cu nivelul de bază (*entry*), dar, după declaraţiile producătorului, prezenta şi multe elemente suplimentare specifice celorlalte două nivele superioare. Certificarea nivelului de conformitate cădea în sarcina unui organism independent, *National Institute for Standards and Technology* (NIST), care utiliza *Federal Information Processing Standards* (FIPS), FIPS PUB 127-2. Însă din 1997, FIPS şi-a declinat implicarea în activitatea de certificare (vezi şi [Gorman 1997]).

Despre SQL-92, referit uneori şi ca SQL2, se spune că este cel mai important dintre standardele SQL, ba chiar cel mai important dintre standardele dedicate bazelor de date.

3.1.4. SQL:1999

În 1997, grupul X3 al ANSI a fost redenumit în NCITS (National Committee for Information Technology Standards), iar comitetul care se ocupă de standardizarea SQL se numeşte acum *ANSI NCITS-H2 (Database)*.

Următorul standard a fost numit iniţial SQL3. Cum cea mai mare parte a sa a fost publicată în iulie 1999, denumirea propusă de ANSI şi ISO este SQL:1999. Complexitatea superioară faţă de predecesori este sugerată şi de numărul de pagini, aproape 2.000 (faţă de 600 ale SQL-92). Scadenţa finalizării sale a fost repetat amânată. Principalele orientări ale SQL:1999 au vizat transformarea acestuia într-un limbaj complet, în vederea definirii şi gestionării obiectelor complexe şi persistente. Aceasta include:

- generalizare şi specializare;
- moşteniri multiple;
- polimorfism;
- încapsulare;
- tipuri de date definite de utilizator;
- expresii privind interogări recursive şi instrumente adecvate de administrare a datelor.

Defalcarea iniţială (1993) a standardului SQL operată de comitetele ANSI şi ISO a avut în vedere şapte componente[5]:

- Partea 1. *Cadrul general* descrie fiecare parte a standardului şi conţine informaţii comune tuturor părţilor.
- Partea 2. *Fundamentul* defineşte sintaxa şi semantica SQL în ceea ce priveşte definirea şi manipularea bazei de date, inclusiv opţiuni privind gestiunea tipurilor abstracte de date.
- Partea 3. *SQL/CLI* (*Call Level Interface*) - un ansamblu de funcţii şi proceduri pentru conectarea bazelor de date prin SQL în medii multiutilizator şi multi-platformă, ansamblu dezvoltat de SQL Access Group. Cea mai faimoasă implementare a SQL/CLI este ODBC (Object DataBase Connectivity), la care firma Microsoft a avut o contribuţie decisivă.
- Partea 4. *SQL/PSM* (*Persistent Stored Modules*) - specificaţii procedurale necesare în funcţii şi proceduri utilizator.
- Partea 5. *SQL/Bindings* include *Dynamic SQL* şi *Embedded SQL* din standardul SQL92 (SQL2) care se referă la modul cum SQL este inclus în limbajele de programare neobiectuale. SQL/OLB (Object Language Bindings) furnizează posibilitatea de a include comenzi SQL în programe Java, pe baza JDBC (Java DataBase Connectivity).
- Partea 6. *SQL/XA* - specificaţii elaborate de X/Open şi dedicate platformei X Windows. Această parte a fost abandonată.
- Partea 7. *SQL/Temporal* adaugă noi facilităţi privind gestiunea timpului şi datei calendaristice în SQL.

5. http://www.jcc.com/SQLPages/jccs_sql.htm.

La această structurare iniţială, între timp au mai fost adăugate şi alte părţi precum[6]:

- *SQL/OLAP* - sunt descrise funcţiile şi operaţiunile utilizate pentru prelucrări analitice, fiind publicate ca amendament la standardul SQL:1999.
- Partea 8. *SQL/Objects - Extended Objects* vizează modul în care SGBD-urile relaţionale gestionează tipurile abstracte de date în aplicaţii. Nici această componentă nu mai există astăzi, fiind transferată în întregime în Fundament.
- Partea 9. *SQL/MED (Management of External Data)* defineşte câteva elemente adiţionale Fundamentului pentru accesarea unor surse de date de tipul fişierelor şi bazelor de date.
- Partea 10. *SQL/OLB (Object Language Bindings)* este inclus numai în standardul ANSI şi definitivat din 1998; cuprinde specificaţii privitoare la includerea frazelor SELECT în limbajul Java, fiind corespondentă unui alt standard ANSI, SQLJ (Partea 0).
- Partea 11. *SQL/Schemata* se referă la definirea şi extragerea informaţiilor privind schema bazei de date; este parte din Fundament, dar, în viitor, această parte va fi de sine stătătoare.
- *SQL Routines using the Java Programming Language.* Definitivată în 1999 şi inclusă numai în standardul ANSI, această parte se referă la modul în care secvenţe de cod Java pot fi incluse în bazele de date SQL.
- Partea 12. *SQL/Replication* - lucrările au demarat în 2000, fiind vizată standardizarea comenzilor pentru replicarea bazelor de date.

Dintre ameliorările nelegate strict de lucrul cu obiecte, merită amintite cele referitoare la: declanşatoare (*triggere*) şi alte tipuri de proceduri stocate, suport pentru seturile de caractere naţionale, noi predicate în clauza WHERE (FOR ALL, FOR SOME, SIMILAR TO), tabele virtuale actualizabile, roluri pentru definirea profilelor de securitate etc. Se poate spune că, odată cu standardul publicat în 1999, SQL iese din sfera relaţionalului.

3.1.5. SQL:2003

Dominanta standardului SQL:2003 este XML. În afară de corecţia câtorva inconsecvenţe ale SQL:1999, au mai fost:

- eliminate tipurile de date BIT şi BIT VARYING;
- introduse tipurile de date BIGINT, MULTISET, XML;
- modificate câteva opţiuni DDL (CREATE TABLE LIKE şi CREATE TABLE AS) şi DML (MERGE);
- introduse câteva noi opţiuni, cum ar fi funcţiile-tabelă, secvenţe, coloane-identificator (vezi şi [Eisenberg *et al.* 2004]).

SQL:2003 cuprinde nouă părţi, opt preluate din SQL:1999, plus una dedicată XML-ului ([Kriegel & Trukhnov 2008], p. 25, [Kulkarni 2003]):

6. Adaptat după [Hernandez & Viescas 2000], pp. 62-63.

- Cadru general - *SQL Framework* (partea 1);
- Fundament - *SQL Foundation* (partea 2);
- *SQL/CLI* (Call Level Interface) (partea 3);
- *SQL/PSM* (Persistent Stored Modules) (partea 4);
- *SQL/MED* (Management of External Data) (partea 9);
- *SQL/OLB* (Object Language Bindings) (partea 10);
- *SQL/Schemata* (partea 11);
- *SQL/JRT* (Java Routines and Types) (partea 13);
- *SQL/XML* (partea 14).

3.1.6. SQL:2008

Pe site-ul ISO a fost publicat (şi poate fi cumpărat) SQL:2008[7], elaborat, în colaborare, de ISO şi IEC (International Electrotechnical Commission). Cele două organisme internaţionale au o comisie tehnică comună, dedicată exclusiv standardelor ce privesc tehnologiile informaţionale - *JTC 1* (Joint Technical Committee 1) şi, în cadrul acesteia, subcomisia SC32 denumită *Data management and interchange*. Stardardele se aprobă dacă există o rată de acceptare de cel puţin 75% dintre organismele naţionale de standardizare.

Asemenea standardului precedent (ISO/IEC 9075:2003), versiunea ISO/IEC 9075:2008 prezintă nouă părţi, cu titulaturi identice. Noutăţile vizează partea dedicată XML, adăugarea unui nou tip BINARY, suport pentru căutarea pe bază de şabloane (*regular expressions*, vezi capitolul 5), clasamente (FIRST n/TOP n) etc. ([Kriegel & Trukhnov 2008], p. 25).

Nu aş vrea să credeţi că standardizarea SQL este un marş triumfal. Lumea teoreticienilor şi profesioniştilor în ale bazelor de date este destul de fragmentată în aprecierea virtuţilor şi slăbiciunilor stardardelor SQL. Obiecţiile sunt atât de natură conceptuală, venind dinspre tabăra teoreticienilor relaţionali (E.F. Codd, C.J. Date, H. Darwen, F. Pascal), dar şi dinspre practicieni care deplâng lipsa de conformitate SQL a serverelor BD actuale faţă de ultimele două-trei versiuni ale standardului ([Gulutzan 2005], [Gorman 2001]).

3.2. Tipuri de date primitive SQL

La crearea „efectivă" a oricărei tabele dintr-o bază gestionată de un server BD, pentru fiecare atribut, pe lângă nume şi eventuale restricţii trebuie declarat tipul acestuia - dacă valorile sale vor fi numere, şiruri de caractere etc. Mult timp, modelului relaţional i s-a imputat că poate gestiona numai date simple, adică numere, şiruri de caractere şi date calendaristice. Primele două standarde SQL erau „puse la index" pe acelaşi motiv. Începând cu SQL:1999, tipologia datelor este mult mai generoasă. În afară de tipurile predefinite noi, precum INTERVAL sau

7. www.ansi.org.

ARRAY, au fost introduse opţiuni care să le permită utilizatorilor să-şi definească şi să-şi implementeze propriile tipuri, în funcţie de cerinţele aplicaţiilor.

SQL:2008-1 defineşte trei tipuri de date: tipuri *predefinite*, tipuri *construite* şi tipuri *definite de utilizator*. În următoarele capitole, ne vom opri cu precădere asupra tipurilor din prima categorie, pe care le putem denumi *primitive* sau *clasice*, deşi în discutarea lor vor exista câteva trimiteri şi la tipuri din celelalte două categorii. Vom trece sub tăcere datele de tip XML, deşi sunt la modă.

3.2.1. Categoriile „clasice" de tipuri de date

Deşi există unele diferenţe între dialectele SQL-urile, următoarele categorii de tipuri de date sunt cvasiprezente încă din primele două standarde:

- pentru numere:
  - SMALLINT: întregi - scurte,
  - INTEGER sau INT: întregi,
  - NUMERIC(p,s) sau DECIMAL(p,s) sau DEC(p,s) - reale, cu un total de p poziţii, din care s la partea fracţionară,
  - FLOAT: reale, virgulă mobilă,
  - REAL: reale, virgulă mobilă (cu precizie mai mică decât FLOAT, dar la nivelul de intrare este identic),
  - DOUBLE PRECISION: reale, virgulă mobilă, dublă precizie;
- pentru şiruri de caractere:
  - CHAR(n) sau CHARACTER(n): şir de caractere de lungime (fixă),
  - VARCHAR(n) sau CHAR VARYING(n) sau CHARACTER VARYING(n): şir de caractere de lungime variabilă;
- pentru date şi ore calendaristice:
  - DATE: dată calendaristică,
  - TIME: oră, minut, secundă, după caz şi fracţiuni de secundă,
  - TIMESTAMP: an, lună, zi, ora, minut, secundă.

3.2.2. Categorii recente de tipuri de date

De la SQL:1999 lucrurile s-au schimbat mult (în bine) în materie de tipuri de date. O serie de tipuri clasice au fost completate cu subtipuri, însă cele mai interesante sunt câteva tipuri nou-nouţe.

Numere

SQL:2003 introduce, printre altele, tipul BIGINT care, după cum îi spune şi numele, permite stocarea şi manipularea unor numere întregi de dimensiuni mai mari decât INTEGER.

Şiruri de caractere

Alături de „clasicele" CHARACTER (CHAR) şi CHARACTER VARYING (VARCHAR), de la SQL:1999 apare tipul CHARACTER LARGE OBJECT (CLOB) util pentru atribute ale căror valori sunt şiruri de caractere de lungime variabilă şi

suficient de mare pentru a fi exprimată în kilo (1.024, mega (1.048.576) şi giga (1.073.741.824) caractere.

Mai merită amintit că toate tipurile pentru şiruri de caractere pot fi gestionate nu numai în engleză, ci şi în alte limbi, unele exotice, din punctul de vedere al caracterelor folosite şi principiilor fonetice (Hebrew, Hindu, Japanese, Chinese, Korean etc.)[8]. Pentru aceasta, standardul SQL:1999 a prevăzut folosirea clauzei NATIONAL, fiind, astfel, disponibile trei (relativ) noi tipuri: NATIONAL CHARACTER (NCHAR), NATIONAL CHARACTER VARYING (NCHAR VARYING) şi NATIONAL CHARACTER LARGE OBJECT (NCLOB). Implicaţiile sunt importante atât pentru stocare, cât şi prelucrare/ordonări.

BLOB-uri

Tipul de date BINARY LARGE OBJECT a fost gândit pentru stocarea în baza de date a imaginilor, sunetelor şi secvenţelor video. Ca şi în cazul CLOB-urilor, lungimea unui atribut de tip BLOB se poate exprima în multipli de octeţi: K, M şi G. Limitele de mărime ale BLOB-urilor diferă de la un SGBD la altul. Principalul dezavantaj al acestui tip este opacitatea sa, SGBD-urile neavând implementate funcţii care să permită extragerea valorilor după anumite criterii. De exemplu, am fi interesaţi ca, dintre toate pozele (atributul FOTOGRAFIE_BULETIN) pe care le avem într-o tabelă destinată persoanelor, să le extragem numai pe cele în care, pe obrazul stâng apare o cicatrice. Lucrul acesta este posibil, în prezent, doar cu un software superspecializat.

Date, ore şi intervale calendaristice

În privinţa datelor temporale, Joe Celko operează o delimitare între [Celko 1999]:

- *evenimente*, pentru care există tipurile DATE (zi, lună, an), TIME (oră, minut, secundă), TIMESTAMP (este o combinaţie DATE + TIME + fracţiuni de ordinul 6 de secundă);
- *intervale* - perioade de timp dintre două evenimente (momente) - tipul INTERVAL (YEAR TO MONTH sau DAY TO HOUR/MINUTE/ SECOND) şi
- *perioade* - intervale cu un moment fix de început, pentru care este necesară o combinaţie de două câmpuri, fie TIMESTAMP - TIMESTAMP, fie TIMESTAMP - INTERVAL.

Fără altă clauză, ora considerată de sistem este UTC (Universal Coordinated Time) sau GMT (Greenwich Mean Time), cum i se spunea anterior. Există însă posibilitatea de a declara şi ora unei alte zone orare (Bucureştiul are ora GMT + 2), aşa încât la tipurile de mai sus mai socotiţi două:

- TIME WITH TIME ZONE;
- TIMESTAMP WITH TIME ZONE;

8. Gestiunea seturilor naţionale de caractere este foarte importantă şi pentru ordonări (Ş urmează după S) sau comparaţii. Primele asemenea opţiuni au fost introduse însă din SQL-92.

Pentru aceste două tipuri, diferenţa faţă de ora UTC se exprimă ca interval de tip HOUR TO MINUTE cu valori de la -12:59 la +13:00[9].

În privinţa intervalelor de date, acestea pot conţine şi ani, şi luni (INTERVAL YEAR TO MONTH), numai ani (INTERVAL YEAR) sau numai luni (INTERVAL MONTH). Dacă mărimea intervalului o cere, se poate specifica numărul de poziţii al anilor (atât pentru INTERVAL YEAR, cât şi INTERVAL YEAR TO MONTH) sau lunilor (numai pentru INTERVAL MONTH). De exemplu, când un interval în care ne interesează deopotrivă anii şi lunile are maximum ordinul sutelor de ani, se recomandă folosirea tipului INTERVAL YEAR(3) TO MONTH. Dacă nu ne-ar interesa decât anii, atunci tipul al fi INTERVAL YEAR(3). Când acelaşi interval trebuie exprimat doar în luni, se foloseşte tipul INTERVAL MONTH(4). În niciun caz nu se poate declara INTERVAL YEAR(2) TO MONTH(3).

Aceeaşi regulă (a specificării numărului de poziţii doar pentru prima dintre mărimile de exprimare a intervalului) se foloseşte şi pentru: INTERVAL DAY(n) TO HOUR, INTERVAL DAY(n) TO MINUTE, INTERVAL DAY(n) TO SECOND, INTERVAL HOUR(n) TO SECOND, INTERVAL MINUTE(n), INTERVAL SECOND(n) etc.

Valori logice

Paradoxal, până în SQL:1999, tabelele nu puteau conţine atribute ale căror valori să fie valorile logice TRUE sau FALSE (plus NULL-ul de rigoare). Şi mai paradoxal, nici acum servere de baze de date din categoria „supergrea" (Oracle şi DB2) nu au această „facilitate". Fireşte, lucrurile nu sunt atât de negre, pentru că soluţii ocolitoare există: atribute de tip NUMBER(1) pentru care se declară o restricţie, astfel încât valorile să fie numai 0 sau 1; atribute de tip CHAR(1) cu singurele valori permise „D" sau „N". Începând cu cel de-al treilea standard, SQL introduce tipul BOOLEAN.

Colecţii

În decursul anilor, au existat multe discuţii pe marginea tipurilor colecţii. Abia în SQL:1999 a fost introdus (doar) tipul ARRAY ([Türker & Gertz 2001], [Melton & Simon 2002], p. 42), la care, în SQL:2003 s-a adăugat MULTISET. Tipul ARRAY este, prin comparaţie cu alte limbaje de programare, unul restrictiv, în sensul că elementele unui masiv nu pot fi, la rândul lor, masive, iar colecţia este unidimensională (vector, şi nu matrice).

Exemplu: *INTEGER ARRAY [10]* specifică un tip colecţie cu maximum 10 elemente, fiecare element fiind un număr întreg. Pe de altă parte, numărul maxim de componente specificat (10) nu înseamnă că pentru fiecare variabilă sau atribut de acest tip vor fi automat rezervate toate cele zece poziţii (lucrurile se prezintă oarecum similar raportului VARCHAR/CHAR). Pentru o variabilă sau un atribut definit pe tipul ARRAY de mai sus, numărul de componente al valorii curente reprezintă

9. Logic părea ca intervalul să fie între -11:59 şi 12:00, însă din cauza orei de vară este necesar ca a doua valoare să fie 13:00. Vezi [Melton & Simon 2002], pp. 39-40.

cardinalitatea curentă, în timp ce 10 este cardinalitatea maximă ([Melton & Simon 2002], p. 43).

În ciuda relativei sărăcii din standardele SQL, fiecare SGBD are propriile tipuri de colecții, mult mai generoase, după cum vom vedea şi în capitolele din partea a doua a lucrării.

Linii „anonime"

Un alt tip consacrat de SQL:1999 este tipul *tuplu* sau tipul *linie „anonimă"*. Discuția pare ciudată, întrucât tuplul este o noțiune fundamentală a modelului relațional. Noutatea ține de faptul că până la SQL:1999 o variabilă sau un parametru nu puteau fi declarate de tip linie, deşi în multe SGBD-uri lucrul acesta era posibil. Formatul este similar, până la un punct, cu declararea atributelor unei tabele sau a unui tip definit de utilizator:

```
ROW (cnp NUMBER(13), nume VARCHAR(50), prenume VARCHAR(50))
```

Referințe

Nu este vorba, după cum probabil v-ați imaginat (ca mine), despre un tip legat de restricții referențiale, ci despre o trimitere la modelul obiectual, de fapt, la modelul relațional-obiectual al cărui portdrapel încearcă SQL să fie. În capitolul 19, vom discuta despre tabele-obiect, în care fiecare linie conține o instanță a unei clase (tip) de obiecte. Or, orientarea pe obiecte a consacrat ideea de *identificator unic al obiectului* (OID) - o valoare ce diferențiază un obiect de oricare altul. Valoarea unică ce diferențiază o linie-obiect dintr-o asemenea tabelă este denumită valoare--referință, iar tipul acesteia tip-referință ([Melton & Simon 2002], pp. 46-47).

Locatori

Aplicațiile complexe ce folosesc baze de date în care apar atribute de tip CLOB sau BLOB, cu dimensiuni „*mega*litice" sau „*giga*litice", ridică importante probleme de performanță (viteză de lucru), dat fiind imensul transfer dintre platforma-client (stația de lucru) şi serverul BD, aflate uneori la o distanță apreciabilă. Tipul locator a fost gândit tocmai pentru dezvoltatorii de aplicații şi mai puțin pentru stocarea datelor în bază. De fapt, locatorul este valabil numai pe stratul interfață al aplicației, fiind imposibil de folosit la definirea tipului atributelor în tabele ([Melton & Simon 2002], p. 47). Există trei categorii de locatori, pentru obiecte mari, pentru tipuri-utilizator şi pentru colecții. Nefiind un tip util stocării datelor în bază, ne oprim aici cu discuția.

Tipuri definite de utilizator

Un pas imens în evoluția SQL făcut de SQL:1999 îl reprezintă libertatea utilizatorilor (mă rog, a celor care au suficiente drepturi sau „privilegii") de a defini şi de a stoca în baza de date a unor tipuri noi, în funcție de cerințele aplicațiilor. Despre această realizare care consacră modelul relațional extins sau relațional-obiectual, vom discuta în capitolul 19.

3.3. Tabele şi restricţii în SQL

Crearea unei baze de date are o componentă tehnică pronunţată, mai ales în cazul serverelor de baze de date: Oracle, DB2, SQL Server, Informix, PostgreSQL, MySQL etc. Chiar dacă există instrumente grafice care uşurează considerabil această activitate, crearea unei baze de date necesită cunoştinţe de administrare, strict legate de produsul software de gestiune a bazei. În continuare, vom eluda elementele tehnice/fizice (spaţiile-tabelă, segmentele de rollback, fişierele de date, procesele-sistem etc.), lăsându-le în seama administratorilor BD[10]; vom considera baza de date creată din punct de vedere fizic, astfel încât ne interesează numai crearea tabelelor, modificarea structurii lor şi declararea restricţiilor.

3.3.1. Crearea tabelelor

Comanda SQL utilizată pentru crearea unei tabele este CREATE TABLE. Precizăm că este vorba despre crearea *structurii* tabelei. Popularea cu înregistrări şi actualizarea ulterioară se realizează prin alte comenzi: INSERT, UPDATE, DELETE. Tabelele sunt similare celei din baza de date VÂNZĂRI. Pentru crearea unei tabele şi declararea atributelor acesteia, fără nicio referire la restricţii, comanda CREATE TABLE are, în cazul tabelei JUDEŢE, următoarea sintaxă:

```
CREATE TABLE judete (
        Jud CHAR(2),
        Judet VARCHAR(25),
        Regiune VARCHAR (15)
   );
```

Întrucât atributele acestei tabele sunt exclusiv de tip şir de caractere, s-au folosit opţiunile CHAR şi VARCHAR. CHAR este preferat pentru atributul Jud, deoarece indicativul auto al judeţului este format exclusiv din două litere (cu excepţia Bucureştiului). Întrucât denumirile judeţului şi regiunii au lungime variabilă, pentru aceste atribute a fost preferat tipul VARCHAR. Pentru ilustrarea şi a altor două tipuri de date, prezentăm comanda de creare a tabelei FACTURI:

```
CREATE TABLE facturi (
        NrFact NUMERIC(8),
        DataFact DATE,
        CodCl DECIMAL(6),
        Obs VARCHAR(50)
   );
```

NrFact şi CodCl sunt de tip numeric, în timp ce DataFact de tip dată calendaristică. Modelul relaţional nu face referire explicită la această noţiune, însă SGBD-urile permit declararea, pentru fiecare atribut, a unei valori implicite (sau valoarea „din oficiu") – DEFAULT. Iată un exemplu:

10. Parcurgerea paşilor necesari creării unei baze de date în Oracle este descrisă în [Fotache *et al.* 2003].

```
CREATE TABLE facturi (
        NrFact NUMERIC(8),
        DataFact DATE DEFAULT CURRENT_DATE,
        CodCl DECIMAL(6) DEFAULT 1001,
        Obs VARCHAR(50)
    );
```

Ca urmare a clauzei DEFAULT, în orice linie adăugată în tabela FACTURI, valoarea implicită (dacă nu este specificată în comanda INSERT) a DataFact va fi data sistemului (data curentă), iar CodCl se va inițializa cu valoarea 1001.

La modul general, când se inserează o linie într-o tabelă (vezi comenzile INSERT din paragraful 3.4), pentru fiecare atribut la care nu se specifică valoarea, aceasta va fi, automat, cea declarată în clauza DEFAULT. Când clauza DEFAULT lipseşte, iar valoarea atributului nu se specifică în comanda INSERT, atributul va „primi" valoarea NULL. Clauza DEFAULT poate conține, pe lângă literali/constante, funcții-sistem de tip dată/timp - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCAL_TIMESTAMP, funcții-sistem de genul CURRENT_USER, SESSION_USER, SYSTEM_USER sau NULL.

Ar mai trebui spus că o tabelă poate fi creată şi pe baza unei interogări, după cum vom vedea la începutul capitolului 13, în care ne vom ocupa şi de crearea tabelelor temporare globale şi locale.

Paragrafele următoare, 3.3.2-3.3.6, ilustrează modul în care pot fi declarate în SQL restricțiile discutate în capitolul precedent.

3.3.2. Restricția de domeniu

Standardele SQL definesc noțiunea de *domeniu*, precum şi comanda pentru declararea unui domeniu - CREATE DOMAIN. Ținând seama că există doar câteva regiuni etnografice în țară, putem defini explicit un domeniu care să conțină valorile „autorizate":

```
CREATE DOMAIN dom_regiuni VARCHAR(20)
        CHECK (VALUE IN ('Dobrogea', 'Moldova', 'Muntenia', 'Oltenia', 'Transilvania'))
```

Dorim ca denumirea unui județ să fie introdusă în baza de date pe un şablon unitar: prima literă din fiecare cuvânt al denumirii județului să fie majusculă, iar restul literelor să fie „minuscule"; în plus, denumirea județului nu trebuie să conțină spații la început. Putem crea un domeniu pentru acest tip de valori:

```
CREATE DOMAIN dom_judet VARCHAR(30)
        CHECK (
        VALUE IS NOT NULL AND VALUE = TRIM (LEADING FROM
        INITCAP(VALUE))
            );
```

La crearea unei tabele - să-i zicem JUDEȚE_TEST, ultimele două atribute vor fi definite pe aceste domenii:

```
CREATE TABLE judete_test (
```

```
jud CHAR(2),
judet dom_judet,
regiune dom_regiuni
);
```

Sintaxa de mai sus funcţionează în puţine SGBD-uri (comenzile prezentate au fost rulate în PostgreSQL). O parte dintre membrii comitetului de standardizare ANSI/INCTIS au afirmat în repetate rânduri că noţiunea de *domeniu* va fi eliminată din standardele următoare (vezi, spre exemplu, [Melton & Simon 2002], p. 98). Este, probabil, unul dintre motivele pentru care SGBD-uri importante (de exemplu, Oracle, DB2, SQL Server) nu lucrează cu domenii de maniera prezentată în standard.

3.3.3. Restricţia de nenulitate

Unele atribute, obligatoriu cele din cheia primară, nu pot prezenta valori nule. Pentru aceasta, la crearea tabelei şi declararea atributului se foloseşte opţiunea NOT NULL:

```
CREATE TABLE facturi (
    NrFact NUMERIC(8) NOT NULL,
    DataFact DATE DEFAULT CURRENT_DATE NOT NULL,
    CodCl DECIMAL(6) DEFAULT 1001 NOT NULL,
    Obs VARCHAR(50)
    );
```

3.3.4. Cheie primară/unicitate

Cheia primară a unei relaţii este definită prin clauza PRIMARY KEY, plasată fie imediat după atributul-cheie, fie după descrierea ultimului atribut al tabelei. A doua variantă este întrebuinţată cu precădere atunci când cheia primară a tabelei este compusă. Pentru atributele de tip cheie-candidat se poate folosi clauza UNIQUE, care va asigura respectarea unicităţii valorilor. Iată câteva variante de folosire a clauzelor PRIMARY KEY şi UNIQUE:

```
CREATE TABLE facturi (
    NrFact NUMERIC(8) NOT NULL PRIMARY KEY,
    DataFact DATE DEFAULT CURRENT_DATE NOT NULL,
    CodCl DECIMAL(6) DEFAULT 1001 NOT NULL,
    Obs VARCHAR(50)
    );
CREATE TABLE judete (
    Jud CHAR(2) PRIMARY KEY,
    Judet VARCHAR(25) NOT NULL UNIQUE,
    Regiune VARCHAR (15)
```

```
        );
CREATE TABLE liniifact (
        NrFact NUMERIC(8) NOT NULL,
        Linie SMALLINT NOT NULL,
        CodPr NUMERIC(6) NOT NULL,
        Cantitate NUMERIC(10) NOT NULL,
        PretUnit NUMBER (12),
        PRIMARY KEY (NrFact, Linie), UNIQUE (NrFact, CodPr)
        );
```

Pentru tabela LINIIFACT, cheia primară este combinaţia de atribute (NrFact, Linie); restricţia de unicitate pentru cuplul (NrFact, CodPr) înseamnă că se interzice ca, pe o factură, un produs să apară repetat.

3.3.5. Restricţii referenţiale

Declararea restricţiilor referenţiale se realizează utilizând clauza FOREIGN KEY. Astfel, pentru stabilirea legăturii LINIIFACT-FACTURI, comanda de creare are acum formatul:

```
CREATE TABLE liniifact (
        NrFact NUMERIC(8) NOT NULL,
        Linie SMALLINT NOT NULL,
        CodPr NUMERIC(6) NOT NULL,
        Cantitate NUMERIC(10) NOT NULL,
        PretUnit NUMBER (12),
        PRIMARY KEY (NrFact, Linie),
        UNIQUE (NrFact, CodPr),
        FOREIGN KEY NrFact REFERENCES facturi(NrFact),
        FOREIGN KEY CodPr REFERENCES produse(CodPr)
        );
```

Echivalentă acesteia este şi sintaxa în care renunţăm la clauza FOREIGN KEY, păstrând doar REFERENCES plasată în dreptul atributelor cheii străine:

```
CREATE TABLE liniifact (
        NrFact NUMERIC(8) NOT NULL REFERENCES facturi (NrFact),
        Linie SMALLINT NOT NULL,
        CodPr NUMERIC(6) NOT NULL REFERENCES produse (CodPr),
        Cantitate NUMERIC(10) NOT NULL, PretUnit NUMBER (12),
        PRIMARY KEY (NrFact, Linie), UNIQUE (NrFact, CodPr)
        );
```

În SQL se poate specifica modul în care va fi păstrată integritatea bazei de date la ştergerea unei linii-părinte sau modificarea unei chei primare ce prezintă înregistrări-copil. Pentru a interzice ştergerea unor facturi (înregistrări din

FACTURI) pentru care există tupluri corespondente în LINIIFACT şi, pe de altă parte, pentru ca la modificarea unui număr de factură (NrFact) în FACTURI să se modifice automat, în cascadă, toate liniile-copil din LINIIFACT, formatul comenzii se schimbă astfel:

```
CREATE TABLE liniifact (
        NrFact NUMERIC(8) NOT NULL,
        Linie SMALLINT NOT NULL,
        CodPr NUMERIC(6) NOT NULL,
        Cantitate NUMERIC(10) NOT NULL,
        PretUnit NUMBER (12),
        PRIMARY KEY (NrFact, Linie),
        UNIQUE (NrFact, CodPr),
        FOREIGN KEY NrFact REFERENCES facturi (NrFact)
                ON DELETE RESTRICT ON UPDATE CASCADE,
        FOREIGN KEY CodPr REFERENCES produse (CodPr)
                ON DELETE RESTRICT ON UPDATE CASCADE
        );
```

Există diferenţe semnificative între serverele BD în privinţa definirii acţiunilor ce trebuie întreprinse pentru respectarea integrităţilor referenţiale. Unele, conforme standardului, permit, la declararea restricţiilor referenţiale (este de obicei simultană cu crearea tabelelor), instituirea oricărei reguli din cele patru: ON UPDATE CASCADE, ON UPDATE RESTRICT, ON DELETE CASCADE, ON DELETE RESTRICT.

Altele, precum Oracle sau DB2, permit declararea acţiunilor numai pentru ştergere, ON DELETE CASCADE şi ON DELETE RESTRICT. Pentru aceste servere, modificarea în cascadă a valorilor cheilor primare în înregistrările-copil este posibilă doar prin declanşatoare (vezi capitolul 17).

3.3.6. Restricţii-utilizator

Restricţiile-utilizator, denumite şi restricţii de comportament, sunt implementate de obicei sub forma regulilor de validare la nivel de câmp (*field validation rule*), la nivel de înregistrare (*record validation rule*) sau, eventual, pot fi incluse în declanşatoare (*triggere*). Cu rezerva că aceste reguli pot avea forme complexe şi întrebuinţa elemente avansate de SQL şi/sau extensiile procedurale ale SQL în mediul respectiv, prezentăm în continuare o regulă de validare pentru atributul DataFact în tabela FACTURI şi o alta pentru atributul Linie în LINIIFACT.

```
CREATE TABLE facturi (
        NrFact NUMERIC(8) nn_facturi_nrfact NOT NULL
                CONSTRAINT pk_facturi PRIMARY KEY,
        DataFact DATE DEFAULT CURRENT_DATE
                CONSTRAINT nn_datafact NOT NULL
                CONSTRAINT ck_datafact CHECK (
```

```
        DataFact >= DATE'2007-08-01' AND DataFact <= DATE'2015-12-31'),
    CodCl DECIMAL(6) DEFAULT 1001
    CONSTRAINT nn_facturi_codcl NOT NULL
    CONSTRAINT fk_facturi_clienti REFERENCES CLIENTI(CodCl)
    ON DELETE RESTRICT ON UPDATE CASCADE,
    Obs VARCHAR(50)
    );
CREATE TABLE liniifact (
    NrFact NUMERIC(8) CONSTRAINT nn_liniifact_nrfact NOT NULL,
    Linie SMALLINT CONSTRAINT nn_linie NOT NULL
        CONSTRAINT ck_linie CHECK (Linie > 0),
    CodPr NUMERIC(6) CONSTRAINT nn_liniifact_codpr NOT NULL,
    Cantitate NUMERIC(10) CONSTRAINT nn_cantitate NOT NULL,
    PretUnit NUMERIC (12)
        CONSTRAINT nn_pretunit NOT NULL,
        CONSTRAINT pk_liniifact PRIMARY KEY (NrFact, Linie),
    CONSTRAINT un_liniifact UNIQUE (NrFact, CodPr),
    CONSTRAINT fk_liniifact_facturi FOREIGN KEY NrFact
        REFERENCES  facturi (NrFact)
            ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT fk_liniifact_produse FOREIGN KEY CodPr
        REFERENCES  produse (CodPr)
            ON DELETE RESTRICT ON UPDATE CASCADE
    );
```

Ca urmare a clauzei CHECK, data unei facturi nu poate avea valori în afara intervalului 1 august 2007 - 31 decembrie 2015, iar atributul Linie trebuie să prezinte valori întregi mai mari decât 0.

Ca un alt element de noutate, fiecărei restricţii (cheie primară, unicitate, regulă la nivel de atribut etc.) i s-a dat un nume, lucru util atunci când, la un eveniment de genul salvare, restaurare, încărcarea BD, vrem să dezactivăm una sau mai multe dintre acestea. Pentru a uşura lucrul, se prefixează numele fiecărei restricţii cu tipul său:

- pk_ (PRIMARY KEY) pentru cheile primare;
- un_ (UNIQUE) pentru cheile alternative;
- nn_ (NOT NULL) pentru atributele obligatorii (ce nu pot avea valori nule);
- ck_ (CHECK) pentru reguli de validare la nivel de atribut;
- fk_ (FOREIGN KEY) pentru cheile străine.

3.3.7. Modificarea structurii unei tabele

Odată creată o tabelă, schema sa rămâne constantă. Dacă totuşi, ulterior, se doreşte introducerea, modificarea sau ştergerea unor atribute, sau declararea/anularea unor restricţii, comanda necesară este ALTER TABLE. Câteva exemple:

- pentru adăugarea atributului DataNast (data naşterii) în tabela PERSOANE:

ALTER TABLE PERSOANE ADD COLUMN DataNast DATE

- ştergerea unui atribut:

ALTER TABLE PERSOANE DROP COLUMN DataNast

- modificarea tipului/lungimii unui atribut:

ALTER TABLE PERSOANE ALTER COLUMN Nume VARCHAR(21)

- adăugarea/modificarea valorii implicite a unui atribut:

ALTER TABLE PERSOANE ALTER COLUMN Sex SET DEFAULT 'F'

- anularea unei valori implicite:

ALTER TABLE PERSOANE ALTER COLUMN Sex DROP DEFAULT

- instituirea restricţiei de nenulitate:

ALTER TABLE PERSOANE ADD CONSTRAINT nn_sex Sex IS NOT NULL

- invers, acceptarea valorilor NULL:

ALTER TABLE PERSOANE DROP CONSTRAINT nn_sex

- adăugarea unei reguli de validare la nivel de atribut:

ALTER TABLE PERSOANE ADD CONSTRAINT ck_sex CHECK (sex IN ('M', 'F'))

- ştergerea unei reguli de validare la nivel de atribut:

ALTER TABLE PERSOANE DROP CONSTRAINT ck_sex

Toate restricţiile: cheie primară - PRIMARY KEY, unicitate - UNIQUE, referenţială - FOREIGN KEY, de comportament - CHECK pot fi declarate ulterior creării tabelei şi, bineînţeles, anulate la un moment dat. Spre exemplu, formatul general al comenzii pentru dezactivarea cheii primare este:

ALTER TABLE persoane DROP PRIMARY KEY.

3.3.8. Ştergerea tabelelor

Tabelele pot fi şterse prin comanda DROP TABLE. Dacă, însă, tabela este una părinte (adică are copii), ştergerea trebuie făcută împreună cu toţi copiii (DELETE CASCADE), altminteri SGBD-ul se va împotrivi (va afişa un mesaj de încălcare a restricţiei referenţiale). De multe ori, scripturile de creare a tabelelor unei baze de date (succesiunea comenzilor CREATE TABLE) încep cu DROP TABLE, tocmai pentru a evita erorile cauzate de încercarea de a crea o tabelă care există deja în schema respectivă. La prima execuţie (sau prima lansare de după o execuţie incompletă) a scriptului, este posibil ca o comandă DROP TABLE să genereze o eroare (deoarece se încearcă ştergerea unei tabele inexistente). Lucrul acesta poate fi evitat elegant în PostgreSQL prin folosirea sintaxei *DROP IF EXISTS judete*. În DB2, SQL Server şi Oracle trebuie consultat dicţionarul de date pentru a vedea dacă tabela există, formatul general fiind ceva mai complicat.

3.3.9. Scripturi de creare a tabelelor în DB2

Dialectul SQL din DB2 este destul de aproape de standardele SQL (unii ar spune că şi reciproca este valabilă). Tipurile de date predefinite „clasice" se regăsesc întocmai în DB2. Iată care ar fi scriptul de creare a celor patru tabele din BD TRIAJ (listing 3.1):

Listing 3.1. Script DB2 de creare a tabelelor BD TRIAJ

```
DROP TABLE triaj;
DROP TABLE pacienti;
DROP TABLE garzi;
DROP TABLE doctori;

CREATE TABLE doctori (
    iddoctor SMALLINT NOT NULL
        GENERATED ALWAYS AS IDENTITY (START WITH 1
                INCREMENT BY 1 MINVALUE 1 NO MAXVALUE)
        CONSTRAINT pk_doctori PRIMARY KEY,
    numedoctor VARCHAR(50) NOT NULL,
    specialitate VARCHAR(40),
    datanasterii DATE
    );

CREATE TABLE garzi (
    iddoctor SMALLINT NOT NULL REFERENCES doctori (iddoctor),
    inceput_garda TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    sfarsit_garda TIMESTAMP,
    CONSTRAINT pk_garzi PRIMARY KEY (iddoctor, inceput_garda)
    );

CREATE TABLE pacienti (
    idpacient INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
                (START WITH 1 INCREMENT BY 1 MINVALUE 1 NO MAXVALUE)
        CONSTRAINT pk_pacienti PRIMARY KEY,
    numepacient VARCHAR(60),
    CNP NUMERIC(13),
    adresa VARCHAR(100),
    loc VARCHAR(30),
    judet CHAR(2),
    tara VARCHAR(30) DEFAULT 'Romania',
    serie_nr_act_identitate VARCHAR(20)
    );

CREATE TABLE triaj (
    idexaminare BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
        (START WITH 1 INCREMENT BY 1 MINVALUE 1 NO MAXVALUE),
    dataora_examinare TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    idpacient NUMERIC(10) NOT NULL CONSTRAINT fk_internari_pacienti
        REFERENCES pacienti (idpacient),
    simptome VARCHAR(500),
    tratament_imediat VARCHAR(500),
    sectie_destinatie VARCHAR(30)
    );
```

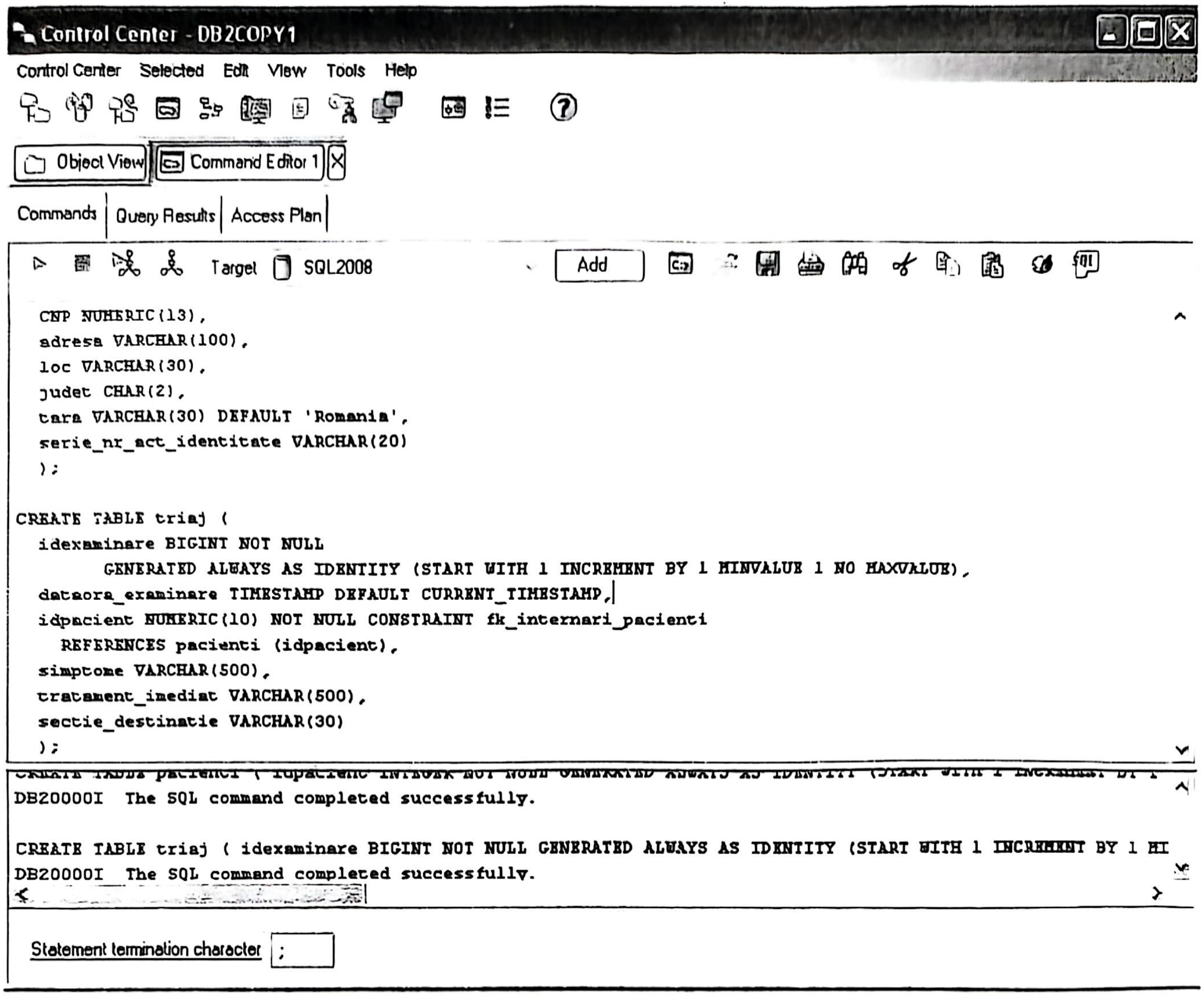


Figura 3.2. Lansarea în execuție în DB2 a scriptului (comenzilor) de creare a tabelelor BD TRIAJ

Lansarea în execuție a acestui script se face din Control Center, folosind, din meniu, opțiunea *Command Editor* - vezi figura 3.2.

Pentru atributul IdDoctor din tabela DOCTORI am ales tipul SMALLINT ale cărui valori minimă şi maximă sunt (în DB2) -32768 şi 32767. În schimb, pentru IdPacient tipul ales a fost INTEGER (pentru, că, în decursul anilor/deceniilor, numărul pacienților poate fi de ordinul sutelor de mii, poate chiar milioanelor (deşi e cam exagerat!)), în timp ce pentru IdExaminare (tabela TRIAJ) tipul ales a fost (tot exagerat) BIGINT. Valorile minime şi maxime ale tipurilor INTEGER şi BIGINT sunt în DB2 -2147483648 şi 2147483647, respectiv (aproximativ) $\pm$ 9,2 * 10^{18}.

IdDoctor, IdPacient şi IdExaminare sunt chei primare în tabelele DOCTORI, PACIENȚI şi TRIAJ. Neavând o semnificație anume, valorile lor pot fi generate automat (1, 2, 3,...), folosind opțiunea GENERATES ALWAYS AS IDENTITY..., ceea ce am şi făcut. Vom vedea, peste un număr de pagini, că aceste coloane autogenerate au un anumit regim la popularea şi modificarea înregistrărilor din tabelele în care apar.

În privința scriptului de creare a tabelelor bazei de date VÂNZĂRI (listingul 3.2), vom încerca, la fel ca şi în viitoarele scripturi dedicare celorlalte trei servere

BD, să implementăm cât mai multe dintre restricţiile definite în paragraful 2.3.2, subliniind eventualele particularităţi. Spre exemplu, în DB2 nu există funcţia INITCAP şi nici alta echivalentă care să ne permită verificarea majusculei fiecărui cuvânt dintr-un şir de caractere. De asemenea, la regulile de validare pentru atributele DataFact din tabela FACTURI, DataInc şi DataDoc din tabela INCASFACT, constantele de tip dată calendaristică se scriu „direct" (*datafact >= '2007-01-01'*), şi nu conform standardului SQL (*datafact >= DATE '2007-01-01'*). Se mai poate folosi însă şi formatul *DATE('2007-01-01')*.

Listing 3.2. Script DB2 de creare a tabelelor BD VÂNZĂRI

```
DROP TABLE incasfact;
DROP TABLE incasari;
DROP TABLE liniifact;
DROP TABLE facturi;
DROP TABLE produse;
DROP TABLE persclienti;
DROP TABLE persoane;
DROP TABLE clienti;
DROP TABLE coduri_postale;
DROP TABLE judete;

CREATE TABLE judete (
    jud CHAR(2) NOT NULL CONSTRAINT pk_judete PRIMARY KEY
        CONSTRAINT ck_jud CHECK (jud=LTRIM(UPPER(jud))),
    judet VARCHAR(25)
        CONSTRAINT un_judet UNIQUE CONSTRAINT nn_judet NOT NULL,
    regiune VARCHAR(15) DEFAULT 'Moldova'
        CONSTRAINT nn_regiune NOT NULL
        CONSTRAINT ck_regiune CHECK (regiune IN ('Banat', 'Transilvania',
                'Dobrogea',  'Oltenia', 'Muntenia', 'Moldova'))
    );

CREATE TABLE coduri_postale (
    codpost CHAR(6) NOT NULL
        CONSTRAINT pk_coduri_post PRIMARY KEY
        CONSTRAINT ck_codpost CHECK (codpost = LTRIM(codpost)),
    loc VARCHAR(25) CONSTRAINT nn_loc NOT NULL,
    jud CHAR(2) DEFAULT 'IS'
                CONSTRAINT fk_coduri_post_jud REFERENCES judete(jud)
    );

CREATE TABLE clienti (
    codcl SMALLINT NOT NULL
        GENERATED ALWAYS AS IDENTITY (START WITH 1001
                INCREMENT BY 1 MINVALUE 1 NO MAXVALUE)
        CONSTRAINT pk_clienti PRIMARY KEY
        CONSTRAINT ck_codcl CHECK (codcl > 1000),
    dencl VARCHAR(30) NOT NULL
                CONSTRAINT ck_dencl CHECK (SUBSTR(dencl,1,1) =
                UPPER(SUBSTR(dencl,1,1))),
    codfiscal CHAR(9) NOT NULL
        CONSTRAINT un_codfiscal UNIQUE
        CONSTRAINT ck_codfiscal CHECK (SUBSTR(codfiscal,1,1) =
                UPPER(SUBSTR(codfiscal,1,1))),
    adresa VARCHAR(40) CONSTRAINT ck_adresa_clienti
CHECK (SUBSTR(adresa,1,1) = UPPER(SUBSTR(adresa,1,1))),
```

```
    codpost CHAR(6) NOT NULL
        CONSTRAINT fk_clienti_cp REFERENCES coduri_postale(codpost),
    telefon VARCHAR(10)
    );

CREATE TABLE persoane (
    cnp CHAR(14) NOT NULL
        CONSTRAINT pk_persoane PRIMARY KEY
        CONSTRAINT ck_cnp CHECK (cnp = LTRIM(UPPER(cnp))),
    nume VARCHAR(20) NOT NULL,
    prenume VARCHAR(20) NOT NULL,
    adresa VARCHAR(50)
        CONSTRAINT ck_adresa_persoane CHECK (SUBSTR(adresa,1,1) =
        UPPER(SUBSTR(adresa,1,1))),
    sex CHAR(1) DEFAULT 'M' NOT NULL
        CONSTRAINT ck_sex CHECK (sex IN ('F','M')),
    codpost CHAR(6) NOT NULL
        CONSTRAINT fk_persoane_cp REFERENCES coduri_postale(codpost),
    telacasa VARCHAR(10),
    telbirou VARCHAR(10),
    telmobil VARCHAR(10),
    email VARCHAR(30)
    );

CREATE TABLE persclienti (
    cnp CHAR(14) NOT NULL
        CONSTRAINT fk_persclienti_persoane REFERENCES persoane(cnp),
    codcl SMALLINT NOT NULL
        CONSTRAINT fk_persclienti_clienti REFERENCES clienti(codcl),
    functie VARCHAR(25) NOT NULL
        CONSTRAINT ck_functie CHECK (SUBSTR(functie,1,1) =
                UPPER(SUBSTR(functie,1,1))),
    CONSTRAINT pk_persclienti PRIMARY KEY (cnp, codcl, functie)
    );

CREATE TABLE produse (
    codpr SMALLINT NOT NULL
        GENERATED ALWAYS AS IDENTITY (START WITH 1
                INCREMENT BY 1 MINVALUE 1 NO MAXVALUE)
        CONSTRAINT pk_produse PRIMARY KEY
        CONSTRAINT ck_codpr CHECK (codpr > 0),
    denpr VARCHAR(30) NOT NULL CONSTRAINT ck_denpr
        CHECK (SUBSTR(denpr,1,1) = UPPER(SUBSTR(denpr,1,1))),
    um VARCHAR(10) NOT NULL,
    grupa VARCHAR(15) CONSTRAINT ck_produse_grupa
        CHECK (SUBSTR(grupa,1,1) = UPPER(SUBSTR(grupa,1,1))),
    procTVA NUMERIC(2,2) DEFAULT .19 NOT NULL
    );

CREATE TABLE facturi (
    nrfact INTEGER NOT NULL
                CONSTRAINT pk_facturi PRIMARY KEY,
    datafact DATE DEFAULT CURRENT_DATE NOT NULL,
                CONSTRAINT ck_datafact CHECK (datafact >= '2007-01-01'
                        AND datafact <= '2015-12-31'),
    codcl SMALLINT NOT NULL
                CONSTRAINT fk_facturi_clienti REFERENCES clienti(codcl) ,
    obs VARCHAR(50)
    );
```

```
CREATE TABLE liniifact (
    nrfact INTEGER NOT NULL
        CONSTRAINT fk_liniifact_facturi REFERENCES facturi(nrfact),
    linie SMALLINT NOT NULL
        CONSTRAINT ck_linie CHECK (linie > 0),
    codpr SMALLINT NOT NULL
        CONSTRAINT fk_liniifact_produse REFERENCES produse(codpr),
    cantitate NUMERIC(8) NOT NULL,
    pretunit NUMERIC (9,2) NOT NULL,
    CONSTRAINT pk_liniifact PRIMARY KEY (nrfact,linie),
    CONSTRAINT un_liniifact UNIQUE (nrfact,codpr)
    );

CREATE TABLE incasari (
    codinc BIGINT NOT NULL
        GENERATED ALWAYS AS IDENTITY (START WITH 1
                INCREMENT BY 1 MINVALUE 1 NO MAXVALUE)
        CONSTRAINT pk_incasari PRIMARY KEY,
    datainc DATE DEFAULT CURRENT_DATE NOT NULL
        CONSTRAINT ck_datainc CHECK (datainc >= '2007-01-01' AND
                datainc <= '2015-12-31'),
    coddoc CHAR(4) NOT NULL
        CONSTRAINT ck_coddoc CHECK(coddoc = UPPER(LTRIM(coddoc))),
    nrdoc VARCHAR(16) NOT NULL,
    datadoc DATE DEFAULT CURRENT_DATE  NOT NULL
        CONSTRAINT ck_datadoc CHECK (datadoc >= '2007-01-01' AND
                datadoc <= '2010-12-31')
    );

CREATE TABLE incasfact (
    codinc BIGINT NOT NULL
        CONSTRAINT fk_incasfact_incasari REFERENCES incasari(codinc),
    nrfact INTEGER NOT NULL
        CONSTRAINT fk_incasfact_facturi REFERENCES facturi(nrfact),
    transa NUMERIC(11,2) NOT NULL,
    CONSTRAINT pk_incasfact PRIMARY KEY (codinc, nrfact)
    );

ALTER TABLE incasari ADD CONSTRAINT ck_date_incdoc
        CHECK (DataInc >= DataDoc);
```

Cheile primare CodCl (din CLIENŢI), CodPr (PRODUSE) şi CodInc (ÎNCASĂRI) au fost declarate atribute autogenerate. Valorile de start sunt: 1001 pentu CodCl, 1 pentru CodPr şi pentru CodInc. În schimb, NrFact (FACTURI) nu, întrucât firma poate avea anumite intervale de valori pe care le poate folosi pentru numerotarea facturilor, aceste intervale nefiind consecutive (de exemplu, compania numerotează facturile de la 2345678 la 3456789, apoi de la 4567890 la 5800400 etc. Chiar dacă valorile generate pentru NrFact trebuie să fie consecutive, ele nu pot fi autogenerate.

Ultima comandă din script este ALTER TABLE prin care am vrut să demonstrăm că restricţiile pot fi declarate şi ulterior creării tabelei, în cazul nostru fiind vorba despre regula la nivel de înregistrare potrivit căreia, în tabela ÎNCASĂRI,

data documentului de plată trebuie să fie mai mică sau cel mult egală cu data încasării (data intrării banilor în cont/casierie).

3.3.10. Scripturi de creare a tabelelor în Oracle

Şi Oracle este unul dintre servere BD al căror dialect SQL este foarte apropiat de standard. Ca diferenţe de standard şi celelalte servere BD amintim, pentru moment:

- tipul DATE este în Oracle echivalent tipului TIMESTAMP (minus zone orare), întrucât datele de acest tip conţin, pe lângă zi-lună-an şi oră-minut-secundă-fracţiuni de secundă; cu toate acestea, noi vom folosi tipul TIMESTAMP, ca în standard;
- tradiţia Oracle a consacrat tipul VARCHAR2 în loc de VARCHAR, deşi explicaţiile acestei substituiri sunt mai degrabă obscure.
- în Oracle există funcţia INITCAP, care converteşte prima literă a fiecărui cuvânt dintr-un şir de caractere în majusculă.

Listingul 3.3 conţine scriptul de creare a celor patru tabele grupate în BD TRIAJ. Spre deosebire de celelate trei servere BD, Oracle nu prezintă tipurile SMALLINT, INTEGER sau BIGINT. De asemenea, pentru coloanele de tip cheie surogat (IdDoctor, IdPacient, IdExaminare, CodCl, CodPr, CodInc), singurul mecanism care să „automatizeze" valorile acestora este cel bazat pe secvenţe şi declanşatoare, mecanism pe care îl vom prezenta abia în capitolul 17.

Listing 3.3. Script Oracle de creare a tabelelor BD TRIAJ

```
DROP TABLE triaj;
DROP TABLE pacienti;
DROP TABLE garzi;
DROP TABLE doctori;

CREATE TABLE doctori (
    iddoctor NUMBER(8) CONSTRAINT pk_doctori PRIMARY KEY,
    numedoctor VARCHAR2(50) NOT NULL,
    specialitate VARCHAR2(40) NOT NULL,
    datanasterii DATE
    );

CREATE TABLE garzi (
    iddoctor NUMBER(8) NOT NULL REFERENCES doctori (iddoctor),
    inceput_garda TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        NOT NULL,
    sfarsit_garda TIMESTAMP,
        CONSTRAINT pk_garzi PRIMARY KEY (iddoctor, inceput_garda)
    );

CREATE TABLE pacienti (
    idpacient NUMBER(10) CONSTRAINT pk_pacienti PRIMARY KEY,
    numepacient VARCHAR2(60),
    CNP NUMBER(13),
    adresa VARCHAR2(100),
    loc VARCHAR2(30),
    judet CHAR(2),
```

```
        tara VARCHAR2(30) DEFAULT 'Romania',
        serie_nr_act_identitate VARCHAR2(20)
        );

    CREATE TABLE triaj (
        idexaminare NUMBER(12)
            CONSTRAINT pk_internari PRIMARY KEY,
        dataora_examinare TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            NOT NULL,
        idpacient NUMBER(10) NOT NULL
            CONSTRAINT fk_internari_pacienti
                REFERENCES pacienti (idpacient),
        simptome VARCHAR2(500) NOT NULL,
        tratament_imediat VARCHAR2(500),
        sectie_destinatie VARCHAR2(30)
        );
```

Lansarea în execuţie a acestor comenzi folosind SQL Developer este ilustrată în figura 3.3. Neavând la dispoziţia tipurile SMALLINT, INTEGER şi BIGINT, dimensiunea atributelor numerice se stabileşte prin cifra/cifrele argument ale tipului NUMERIC. Mie mi se pare un pic ciudat ca un server cu atâta forţă să nu pună la dispoziţie aceste trei tipuri, însă, să recunoaştem, deranjul nu este chiar insuportabil[11].

11. În Oracle există tipuri precum INTEGER, BINARY_INTEGER sau PLS_INTEGER, dar numai în extensia procedurală a SQL care este limbajul PL/SQL.

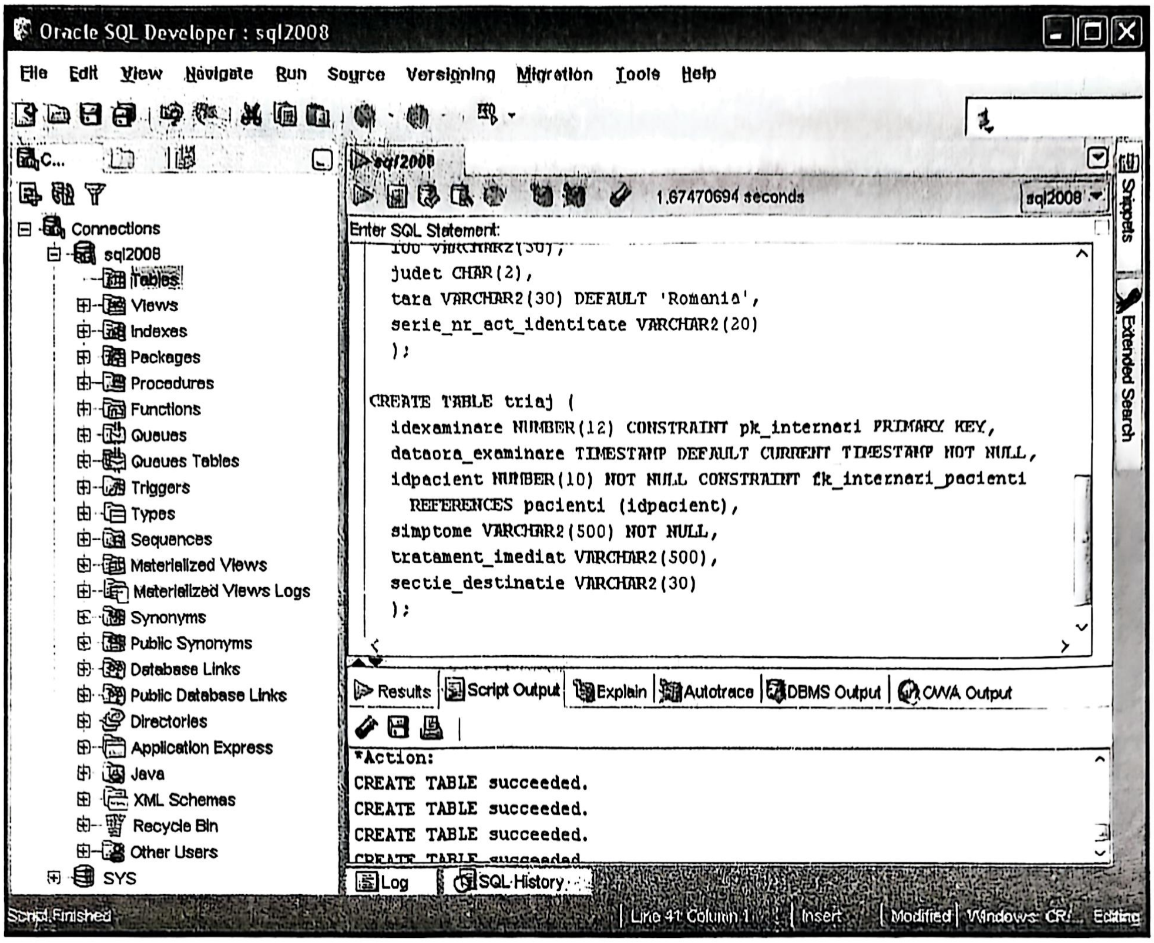


Figura 3.3. Lansarea în execuție în Oracle (SQL Developer) a scriptului (comenzilor) de creare a tabelelor BD TRIAJ

Listingul 3.4 grupează comenzile pentru crearea tabelelor BD VÂNZĂRI. Spre deosebire de varianta DB2, în Oracle în comenzile DROP TABLE, am folosit opțiunea CASCADE CONSTRAINTS care asigură ştergerea unei tabele, chiar dacă mai există în schema BD restricții (cum ar fi cele referențiale) în care tabela respectivă este implicată. De asemenea, clauza DEFAULT a atributului DataDoc din tabela ÎNCASĂRI este expresia CURRENT_DATE - 7[12], neacceptată de DB2.

Listing 3.4 Script Oracle de creare a tabelelor BD VÎNZĂRI

```
DROP TABLE incasfact CASCADE CONSTRAINTS;
DROP TABLE incasari CASCADE CONSTRAINTS;
DROP TABLE liniifact CASCADE CONSTRAINTS;
DROP TABLE facturi CASCADE CONSTRAINTS;
DROP TABLE produse CASCADE CONSTRAINTS;
DROP TABLE persclienti CASCADE CONSTRAINTS;
DROP TABLE persoane CASCADE CONSTRAINTS;
DROP TABLE clienti CASCADE CONSTRAINTS;
DROP TABLE coduri_postale CASCADE CONSTRAINTS;
DROP TABLE judete CASCADE CONSTRAINTS;
```

12. Aceasta ar însemna că, în medie, de la data în care clienții întocmesc documentele de plată până în ziua în care intră banii în contul firmei noastre (şi se operează în baza de date) trec şapte zile, cifră ceva mai mare decât în realitate.

```
CREATE TABLE judete (
    jud CHAR(2)
                CONSTRAINT pk_judete PRIMARY KEY
                CONSTRAINT ck_jud CHECK (jud = LTRIM(UPPER(jud))),
    judet VARCHAR2(25)
                CONSTRAINT un_judet UNIQUE
                CONSTRAINT nn_judet NOT NULL
                CONSTRAINT ck_judet CHECK (judet = LTRIM(INITCAP(judet))),
    regiune VARCHAR2(15)
                DEFAULT 'Moldova' CONSTRAINT nn_regiune NOT NULL
                CONSTRAINT ck_regiune CHECK (regiune IN ('Banat', 'Transilvania',
                 'Dobrogea', 'Oltenia', 'Muntenia', 'Moldova'))
    );

CREATE TABLE coduri_postale (
    codpost CHAR(6)
                CONSTRAINT pk_coduri_post PRIMARY KEY
                CONSTRAINT ck_codpost CHECK (codpost = LTRIM(codpost)),
    loc VARCHAR2(25)
                CONSTRAINT nn_loc NOT NULL
                CONSTRAINT ck_loc CHECK (loc = LTRIM(INITCAP(loc))),
    jud CHAR(2) DEFAULT 'IS'
                CONSTRAINT fk_coduri_post_jud REFERENCES judete(jud)
    );

CREATE TABLE clienti (
    codcl NUMERIC(6)
                CONSTRAINT pk_clienti PRIMARY KEY
                CONSTRAINT ck_codcl CHECK (codcl > 1000),
    dencl VARCHAR2(30) NOT NULL
        CONSTRAINT ck_dencl CHECK (SUBSTR(dencl,1,1) =
                UPPER(SUBSTR(dencl,1,1))),
    codfiscal CHAR(9) NOT NULL
        CONSTRAINT un_codfiscal UNIQUE
        CONSTRAINT ck_codfiscal CHECK (SUBSTR(codfiscal,1,1) =
                UPPER(SUBSTR(codfiscal,1,1))),
    adresa VARCHAR2(40)
        CONSTRAINT ck_adresa_clienti CHECK (SUBSTR(adresa,1,1) =
                UPPER(SUBSTR(adresa,1,1))),
    codpost CHAR(6) NOT NULL
        CONSTRAINT fk_clienti_cp REFERENCES coduri_postale(codpost),
    telefon VARCHAR2(10)
    );

CREATE TABLE persoane (
    cnp CHAR(14)
        CONSTRAINT pk_persoane PRIMARY KEY
        CONSTRAINT ck_cnp CHECK (cnp = LTRIM(UPPER(cnp))),
    nume VARCHAR2(20) NOT NULL
        CONSTRAINT ck_nume CHECK (nume = LTRIM(INITCAP(nume))),
    prenume VARCHAR2(20) NOT NULL
        CONSTRAINT ck_prenume CHECK (prenume = LTRIM(INITCAP(prenume))),
    adresa VARCHAR2(50)
        CONSTRAINT ck_adresa_persoane CHECK (SUBSTR(adresa,1,1) =
                UPPER(SUBSTR(adresa,1,1))),
    sex CHAR(1) DEFAULT 'M' NOT NULL
        CONSTRAINT ck_sex CHECK (sex IN ('F','M')),
    codpost CHAR(6) NOT NULL
        CONSTRAINT fk_persoane_cp REFERENCES coduri_postale(codpost),
```

```
    telacasa VARCHAR(10).
    telbirou VARCHAR(10),
    telmobil VARCHAR(10),
    email VARCHAR2(30)
    );

CREATE TABLE persclienti (
    cnp CHAR(14) NOT NULL
        CONSTRAINT fk_persclienti_persoane REFERENCES persoane(cnp),
    codcl NUMERIC(6) NOT NULL
        CONSTRAINT fk_persclienti_clienti REFERENCES clienti(codcl),
    functie VARCHAR2(25) NOT NULL
        CONSTRAINT ck_functie CHECK (SUBSTR(functie,1,1) =
                UPPER(SUBSTR(functie,1,1))),
        CONSTRAINT pk_persclienti PRIMARY KEY (cnp, codcl, functie)
    );

CREATE TABLE produse (
    codpr NUMERIC(6)
        CONSTRAINT pk_produse PRIMARY KEY
        CONSTRAINT ck_codpr CHECK (codpr > 0),
    denpr VARCHAR2(30) NOT NULL CONSTRAINT ck_denpr
        CHECK (SUBSTR(denpr,1,1) = UPPER(SUBSTR(denpr,1,1))),
    um VARCHAR2(10) NOT NULL,
    grupa VARCHAR2(15) CONSTRAINT ck_produse_grupa
        CHECK (SUBSTR(grupa,1,1) = UPPER(SUBSTR(grupa,1,1))),
    procTVA NUMERIC(2,2) DEFAULT .19 NOT NULL
    );

CREATE TABLE facturi (
    nrfact NUMERIC(8)
        CONSTRAINT pk_facturi PRIMARY KEY,
    datafact DATE DEFAULT CURRENT_DATE NOT NULL,
        CONSTRAINT ck_datafact CHECK (datafact >= DATE'2007-01-01'
                AND datafact <= DATE'2015-12-31'),
    codcl NUMERIC(6) NOT NULL
        CONSTRAINT fk_facturi_clienti REFERENCES clienti(codcl),
    obs VARCHAR2(50)
    );

CREATE TABLE liniifact (
    nrfact NUMERIC(8)
        CONSTRAINT fk_liniifact_facturi REFERENCES facturi(nrfact),
    linie NUMERIC(2) NOT NULL
        CONSTRAINT ck_linie CHECK (linie > 0),
    codpr NUMERIC(6) NOT NULL
        CONSTRAINT fk_liniifact_produse REFERENCES produse(codpr),
    cantitate NUMERIC(8) NOT NULL,
    pretunit NUMERIC (9,2) NOT NULL,
        CONSTRAINT pk_liniifact PRIMARY KEY (nrfact,linie),
        CONSTRAINT un_liniifact UNIQUE (nrfact,codpr)
    );

CREATE TABLE incasari (
    codinc NUMERIC(8)
        CONSTRAINT pk_incasari PRIMARY KEY,
    datainc DATE DEFAULT CURRENT_DATE NOT NULL
        CONSTRAINT ck_datainc CHECK (datainc BETWEEN
                DATE'2007-01-01' AND DATE'2015-12-31'),
```

```
    coddoc CHAR(4) NOT NULL
        CONSTRAINT ck_coddoc CHECK(coddoc = UPPER(LTRIM(coddoc))),
    nrdoc VARCHAR(16) NOT NULL,
    datadoc DATE DEFAULT CURRENT_DATE - 7 NOT NULL
        CONSTRAINT ck_datadoc CHECK (datadoc BETWEEN
                DATE'2007-01-01' AND DATE'2015-12-31')
    );

CREATE TABLE incasfact (
    codinc NUMERIC(8) CONSTRAINT fk_incasfact_incasari REFERENCES incasari(codinc),
    nrfact NUMERIC(8) CONSTRAINT fk_incasfact_facturi REFERENCES facturi(nrfact),
    transa NUMERIC(11,2) NOT NULL,
        CONSTRAINT pk_incasfact PRIMARY KEY (codinc, nrfact)
    );

ALTER TABLE incasari ADD CONSTRAINT ck_date_incdoc CHECK (DataInc >= DataDoc);
```

3.3.11. Scripturi de creare a tabelelor în PostgreSQL

Chiar dacă este un server open source, PostgreSQL este înzestrat cu un dialect SQL redutabil. PostgreSQL gestionează şi tipurile SMALLINT, INTEGER şi BIGINT, ale căror valori minime şi maxime sunt similare celor din DB2. Pentru cheile surogat, există două tipuri, SERIAL (care generează valori de la 1 la 2147483647) şi BIGSERIAL (a cărui valoare maximă generată este cea a tipului BIGINT).

Listing 3.5. Script PostgreSQL de creare a tabelelor BD TRIAJ

```
DROP TABLE IF EXISTS triaj ;
DROP TABLE IF EXISTS pacienti ;
DROP TABLE IF EXISTS garzi ;
DROP TABLE IF EXISTS doctori ;

CREATE TABLE doctori (
    iddoctor SERIAL CONSTRAINT pk_doctori PRIMARY KEY,
    numedoctor VARCHAR(50) NOT NULL,
    specialitate VARCHAR(40) NOT NULL,
    datanasterii DATE
    );

CREATE TABLE garzi (
    iddoctor SMALLINT  NOT NULL
        REFERENCES doctori (iddoctor) ON UPDATE CASCADE,
    inceput_garda TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    sfarsit_garda TIMESTAMP,
    CONSTRAINT pk_garzi PRIMARY KEY (iddoctor, inceput_garda)
    );

CREATE TABLE pacienti (
    idpacient SERIAL CONSTRAINT pk_pacienti PRIMARY KEY,
    numepacient VARCHAR(60),
    CNP NUMERIC(13),
    adresa VARCHAR(100),
    loc VARCHAR(30),
    judet CHAR(2),
    tara VARCHAR(30) DEFAULT 'Romania',
    serie_nr_act_identitate VARCHAR(20)
```

```
    );

CREATE TABLE triaj (
    idexaminare BIGSERIAL CONSTRAINT pk_internari PRIMARY KEY,
    dataora_examinare TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    idpacient INTEGER NOT NULL
        CONSTRAINT fk_internari_pacienti
            REFERENCES pacienti (idpacient) ON UPDATE CASCADE,
    simptome VARCHAR(500) NOT NULL,
    tratament_imediat VARCHAR(500),
    sectie_destinatie VARCHAR(30)
    );
```

Faţă de variantele din DB2 şi Oracle, în listingul 3.5 am folosit opţiunea IF EXISTS a comenzii DROP TABLE, pentru a preveni afişarea mesajelor de eroare la prima lansare în execuţie, atunci când cele patru tabele nu există, iar comenzile DROP nu au ce şterge. Figura 3.4 surprinde momentul lansării în execuţie a scriptului în „clientul" pgAdmin.

Interesant este şi că, dacă în tabela DOCTORI tipul atributului cheie primară (IdDoctor) este SERIAL, acelaşi atribut a fost declarat în tabela-copil (GĂRZI) ca fiind de tip SMALLINT. În schimb, IdPacient a fost declarat de tip SERIAL în PACIENŢI, dar de tip INTEGER în tabela TRIAJ. Altminteri, nu sunt mari diferenţe faţă de listingul Oracle (3.3).

Nici listingul următor - 3.6 - nu este mult diferit faţă de 3.4, dacă facem abstracţie de atributele cu valori autogenerate. Diferenţa principală faţă de scripturile echivalente DB2 şi Oracle ţine de posibilitatea folosirii clauzei ON UPDATE CASCADE la definirea restricţiilor referenţiale, opţiune, paradoxal, inexistentă în cele mai pretenţioase două servere BD.

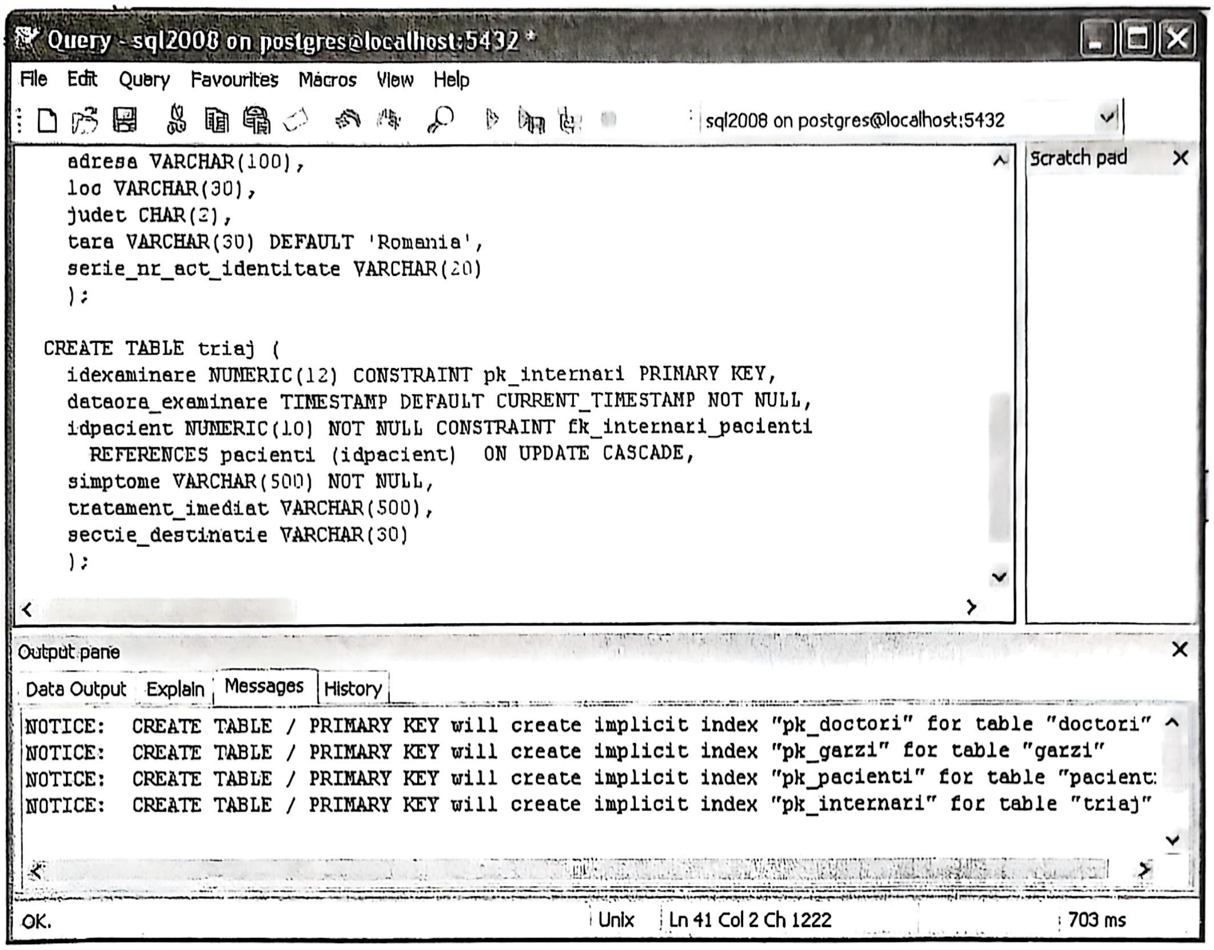


Figura 3.4. Lansarea în execuţie în PostgreSQL (pgAdmin)
a scriptului (comenzilor) de creare a tabelelor BD TRIAJ

Ca şi în Oracle, în PostgreSQL o constantă de tip dată calendaristică (vezi tabelele FACTURI şi ÎNCASĂRI) se pot scrie atât în formatul DATE'an-zi-lună', cât şi cu ajutorul funcţiei TO_DATE[13]. În plus, similar DB2, constantele de tip dată calendaristică pot fi scrise direct: '2008-11-28' (vezi tabela ÎNCASĂRI), deci, în această privinţă avem de unde alege.

Listing 3.6. Script PostgreSQL de creare a tabelelor BD VÂNZĂRI

```
DROP TABLE incasfact;
DROP TABLE incasari;
DROP TABLE liniifact;
DROP TABLE facturi;
DROP TABLE produse;
DROP TABLE persclienti;
DROP TABLE persoane;
DROP TABLE clienti;
DROP TABLE coduri_postale;
DROP TABLE judete;

CREATE TABLE judete (
    jud CHAR(2)
        CONSTRAINT pk_judete PRIMARY KEY
```

13. Vom prezenta detalii despre funcţiile pentru numere, şiruri de caractere şi date/intervale în capitolul 6.

```
        CONSTRAINT ck_jud CHECK (jud = LTRIM(UPPER(jud))),
    judet VARCHAR(25)
        CONSTRAINT un_judet UNIQUE
        CONSTRAINT nn_judet NOT NULL
        CONSTRAINT ck_judet CHECK (judet = LTRIM(INITCAP(judet))),
    regiune VARCHAR(15)
        DEFAULT 'Moldova' CONSTRAINT nn_regiune NOT NULL
        CONSTRAINT ck_regiune CHECK (regiune IN ('Banat', 'Transilvania',
                'Dobrogea', 'Oltenia', 'Muntenia', 'Moldova'))
    );

CREATE TABLE coduri_postale (
    codpost CHAR(6)
        CONSTRAINT pk_coduri_post PRIMARY KEY
        CONSTRAINT ck_codpost CHECK (codpost = TRIM(codpost)),
    loc VARCHAR(25)
        CONSTRAINT nn_loc NOT NULL
        CONSTRAINT ck_loc CHECK (loc = LTRIM(INITCAP(loc))),
    jud CHAR(2) DEFAULT 'IS' CONSTRAINT fk_coduri_post_jud
        REFERENCES judete(jud) ON UPDATE CASCADE
    );

CREATE TABLE clienti (
    codcl SMALLINT NOT NULL
        CONSTRAINT pk_clienti PRIMARY KEY
        CONSTRAINT ck_codcl CHECK (codcl > 1000),
    dencl VARCHAR(30)
        CONSTRAINT ck_dencl CHECK (SUBSTR(dencl,1,1) =
                UPPER(SUBSTR(dencl,1,1))),
    codfiscal CHAR(9)
        CONSTRAINT ck_codfiscal CHECK (SUBSTR(codfiscal,1,1) =
                UPPER(SUBSTR(codfiscal,1,1))),
    adresa VARCHAR(40)
        CONSTRAINT ck_adresa_clienti CHECK (SUBSTR(adresa,1,1) =
                UPPER(SUBSTR(adresa,1,1))),
    codpost CHAR(6)
        CONSTRAINT fk_clienti_cp REFERENCES coduri_postale(codpost)
                ON UPDATE CASCADE,
    telefon VARCHAR(10)
    );

CREATE TABLE persoane (
    cnp CHAR(14)
        CONSTRAINT pk_persoane PRIMARY KEY
        CONSTRAINT ck_cnp CHECK (cnp = LTRIM(UPPER(cnp))),
    nume VARCHAR(20)
        CONSTRAINT ck_nume CHECK (nume = LTRIM(INITCAP(nume))),
    prenume VARCHAR(20)
        CONSTRAINT ck_prenume CHECK (prenume = LTRIM(INITCAP(prenume))),
    adresa VARCHAR(50)
        CONSTRAINT ck_adresa_persoane CHECK (SUBSTR(adresa,1,1) =
                UPPER(SUBSTR(adresa,1,1))),
    sex CHAR(1) DEFAULT 'M'
        CONSTRAINT ck_sex CHECK (sex IN ('F','M')),
    codpost CHAR(6)
        CONSTRAINT fk_persoane_cp REFERENCES coduri_postale(codpost)
                ON UPDATE CASCADE,
    telacasa VARCHAR(10),
    telbirou VARCHAR(10),
```

```
    telmobil VARCHAR(10),
    email VARCHAR(30)
    );

CREATE TABLE persclienti (
    cnp CHAR(14)
        CONSTRAINT fk_persclienti_persoane REFERENCES persoane(cnp)
                ON UPDATE CASCADE,
    codcl SMALLINT
        CONSTRAINT fk_persclienti_clienti REFERENCES clienti(codcl)
                ON UPDATE CASCADE,
    functie VARCHAR(25)
        CONSTRAINT ck_functie CHECK (SUBSTR(functie,1,1) =
                UPPER(SUBSTR(functie,1,1))),
        CONSTRAINT pk_persclienti PRIMARY KEY (cnp, codcl, functie)
    );

CREATE TABLE produse (
    codpr SMALLINT NOT NULL
        CONSTRAINT pk_produse PRIMARY KEY
        CONSTRAINT ck_codpr CHECK (codpr > 0),
    denpr VARCHAR(30) CONSTRAINT ck_denpr
        CHECK (SUBSTR(denpr,1,1) = UPPER(SUBSTR(denpr,1,1))),
    um VARCHAR(10),
    grupa VARCHAR(15) CONSTRAINT ck_produse_grupa
        CHECK (SUBSTR(grupa,1,1) = UPPER(SUBSTR(grupa,1,1))),
    procTVA NUMERIC(2,2) DEFAULT .19
    );

CREATE TABLE facturi (
    nrfact INTEGER
        CONSTRAINT pk_facturi PRIMARY KEY,
    datafact DATE DEFAULT CURRENT_DATE,
        CONSTRAINT ck_datafact CHECK (datafact >= TO_DATE('2007-01-01',
                'YYYY-MM-DD') AND datafact <= DATE'2015-12-31'),
    codcl SMALLINT
        CONSTRAINT fk_facturi_clienti REFERENCES clienti(codcl)
                ON UPDATE CASCADE,
    obs VARCHAR(50)
    );

CREATE TABLE liniifact (
    nrfact INTEGER
        CONSTRAINT fk_liniifact_facturi REFERENCES facturi(nrfact)
                ON UPDATE CASCADE,
    linie SMALLINT CONSTRAINT ck_linie CHECK (linie > 0),
    codpr SMALLINT
        CONSTRAINT fk_liniifact_produse REFERENCES produse(codpr)
                ON UPDATE CASCADE,
    cantitate NUMERIC(8),
    pretunit NUMERIC (9,2),
    CONSTRAINT pk_liniifact PRIMARY KEY (nrfact,linie)
    );

CREATE TABLE incasari (
    codinc BIGINT NOT NULL
        CONSTRAINT pk_incasari PRIMARY KEY,
    datainc DATE DEFAULT CURRENT_DATE
        CONSTRAINT ck_datainc CHECK (datainc >= '2007-01-01'
```

```
                    AND datainc <= TO DATE('31/12/2015','DD/MM/YYYY')),
          coddoc CHAR(4)
               CONSTRAINT ck_coddoc CHECK(coddoc=UPPER(LTRIM(coddoc))),
          nrdoc VARCHAR(16),
          datadoc DATE DEFAULT CURRENT_DATE - 7
               CONSTRAINT ck_datadoc CHECK (datadoc >= DATE'2007-01-01'
                    AND datadoc <= DATE'2010-12-31')
          );

     CREATE TABLE incasfact (
          codinc BIGINT CONSTRAINT fk_incasfact_incasari
               REFERENCES incasari(codinc) ON UPDATE CASCADE,
          nrfact INTEGER CONSTRAINT fk_incasfact_facturi
               REFERENCES facturi(nrfact) ON UPDATE CASCADE,
          transa NUMERIC(11,2) NOT NULL,
               CONSTRAINT pk_incasfact PRIMARY KEY (codinc, nrfact)
          );

     ALTER TABLE incasari ADD CONSTRAINT ck_date_incdoc CHECK (DataInc >= DataDoc);
```

3.3.12. Scripturi de creare a tabelelor în SQL Server

SQL Server a fost (şi încă mai este în oarecare măsură) cel mai puţin fidel standardelor SQL dintre cele patru servere BD pe care le folosim în această carte. Astfel, numai în privinţa datelor de tip întreg, există patru tipuri:

- BIGINT poate lua valori cuprinse între -2^{63} şi $2^{63} - 1$;
- INT poate lua valori cuprinse între -231 şi 231 - 1;
- SMALLINT poate lua valori cuprinse între -215 şi 215 - 1;
- TINYINT poate lua valori cuprinse între 0 şi 255.

În cele patru tabele ale BD TRIAJ am folosit tipul SMALLINT pentru atributul IdDoctor, INTEGER pentru atributul IdPacient şi BIGINT pentru IdExaminare.

Pentru date calendaristice, SQL Server are două tipuri, SMALLDATETIME şi DATETIME. Ambele sunt, de fapt, apropiate de tipul TIMESTAMP din standard. Diferenţa ţine de precizie. În timp ce o valoare SMALLDATETIME are o „precizie" de ordinul minutelor (altfel spus conţine: an, lună, zi, oră, minut), valorile DATETIME au o acurateţe de 3,33 milisecunde (conţin, în plus, secunde şi fracţiuni de milisecunde). Tipul TIMESTAMP are altă funcţionalitate decât în standard şi celelalte servere BD. Versiunea SQL Server 2008 introduce tipul DATETIME, dar noi vom rula aproape toate exemplele în SQL Server 2005.

Listingul 3.7 este echivalentul SQL Server al listingurilor 3.1, 3.3 şi 3.5. Ca tip de dată pentru atributele Început_Gardă şi Sfârşit_Gardă (tabela GĂRZI) am ales tipul SMALLDATETIME, întrucât precizia de ordinul minutelor este suficientă, iar pentru atributul DataOra_Examinare am optat pentru DATETIME.

Atributele pentru care dorim ca valorile să fie unice şi generate automat trebuie declarate în comanda CREATE TABLE cu opţiunea IDENTITY. În lipsa oricărui parametru (cazul nostru), prima valoare generată este 1, iar incrementul este tot 1, valoarea maximă fiind limitată doar de tipul ales (TINYINT, SMALLINT, INTEGER, BIGINT).

Listing 3.7. Script SQL Server de creare a tabelelor BD TRIAJ

```
DROP TABLE triaj;
DROP TABLE pacienti;
DROP TABLE garzi;
DROP TABLE doctori;

CREATE TABLE doctori (
    iddoctor SMALLINT IDENTITY
        CONSTRAINT pk_doctori PRIMARY KEY,
    numedoctor VARCHAR(50) NOT NULL,
    specialitate VARCHAR(40) NOT NULL,
    datanasterii SMALLDATETIME
    );

CREATE TABLE garzi (
    iddoctor SMALLINT NOT NULL
        REFERENCES doctori (iddoctor) ON UPDATE CASCADE,
    inceput_garda SMALLDATETIME
        DEFAULT CURRENT_TIMESTAMP
        NOT NULL,
    sfarsit_garda SMALLDATETIME,
    CONSTRAINT pk_garzi PRIMARY KEY (iddoctor, inceput_garda)
    );

CREATE TABLE pacienti (
    idpacient INTEGER IDENTITY
        CONSTRAINT pk_pacienti PRIMARY KEY,
    numepacient VARCHAR(60) NOT NULL,
    CNP NUMERIC(13),
    adresa VARCHAR(100),
    loc VARCHAR(30),
    judet CHAR(2),
    tara VARCHAR(30) DEFAULT 'Romania',
    serie_nr_act_identitate VARCHAR(20)
    );

CREATE TABLE triaj (
    idexaminare BIGINT IDENTITY CONSTRAINT pk_internari PRIMARY KEY,
    dataora_examinare DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
    idpacient INTEGER NOT NULL
        CONSTRAINT fk_internari_pacienti
                REFERENCES pacienti (idpacient) ON UPDATE CASCADE,
    simptome VARCHAR(500) NOT NULL,
    tratament_imediat VARCHAR(500),
    sectie_destinatie VARCHAR(30)
    );
```

Lansarea în execuție (folosind modulul standard *SQL Server Management Studio*) a comenzilor din acest listing este ilustrată în figura 3.5.

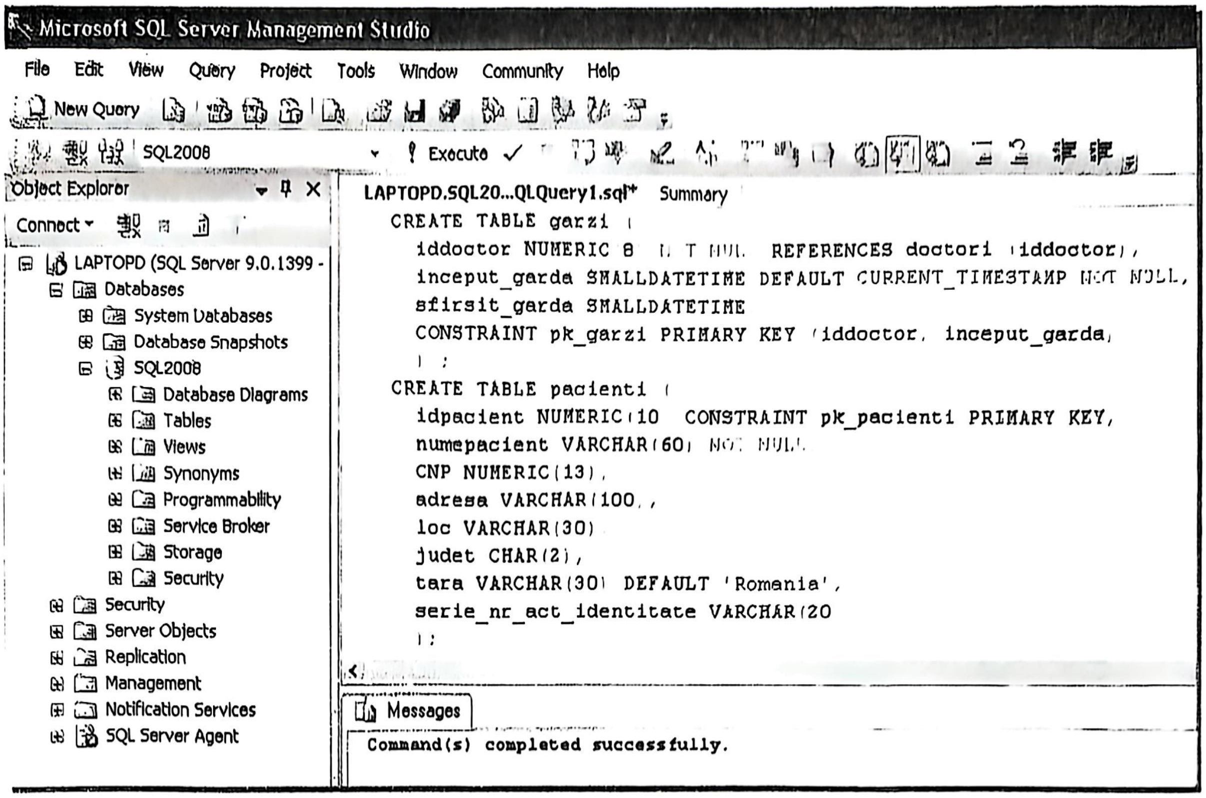


Figura 3.5. Lansarea în execuţie în SQL Server a scriptului (comenzilor) de creare a tabelelor BD TRIAJ

În privinţa comenzilor de (ştergere şi) creare a tabelelor din BD VÂNZĂRI (listing 3.8), am mai adăuga că:

- similar DB2, nu există o funcţie cu funcţionalitatea INITCAP (din Oracle şi PostgreSQL);
- constantele de tip dată calendaristică se scriu ca în DB2, fără „prefixul" DATE;
- pentru atributul Linie din LINIIFACT am folosit tipul TINYINT;
- funcţia care preia data (şi ora) sistemului este GETDATE();
- valoarea de început a atributului CodCl din tabela CLIENŢI este 1001, aşa că la creare am folosit clauza IDENTITY (1001,1); analog am procedat cu atributul CodInc din tabela ÎNCASĂRI, a cărui valoare de start este 1234.

Listing 3.8. Script SQL Server de creare a tabelelor BD VÂNZĂRI

```
DROP TABLE incasfact;
DROP TABLE incasari;
DROP TABLE liniifact;
DROP TABLE facturi;
DROP TABLE produse;
DROP TABLE persclienti;
DROP TABLE persoane;
DROP TABLE clienti;
DROP TABLE coduri_postale;
DROP TABLE judete;

CREATE TABLE judete (
    Jud CHAR(2)
        CONSTRAINT pk_judete PRIMARY KEY
```

```
        CONSTRAINT ck_jud CHECK (Jud = LTRIM(UPPER(Jud))),
    Judet VARCHAR(25)
        CONSTRAINT un_judet UNIQUE
        CONSTRAINT nn_judet NOT NULL,
    Regiune VARCHAR(15)
        DEFAULT 'Moldova' CONSTRAINT nn_regiune NOT NULL
        CONSTRAINT ck_regiune CHECK (Regiune IN ('Banat', 'Transilvania',
                'Dobrogea', 'Oltenia', 'Muntenia', 'Moldova'))
    );

CREATE TABLE coduri_postale (
    CodPost CHAR(6)
        CONSTRAINT pk_coduri_post PRIMARY KEY
        CONSTRAINT ck_codpost CHECK (CodPost = LTRIM(CodPost)),
    Loc VARCHAR(25) CONSTRAINT nn_loc NOT NULL,
    Jud CHAR(2) DEFAULT 'IS'
        CONSTRAINT fk_coduri_post_jud REFERENCES judete(Jud) ON UPDATE CASCADE
    );

CREATE TABLE clienti (
    CodCl SMALLINT IDENTITY (1001,1)
        CONSTRAINT pk_clienti PRIMARY KEY
        CONSTRAINT ck_codcl CHECK (CodCl > 1000),
    DenCl VARCHAR(30) NOT NULL
        CONSTRAINT ck_dencl CHECK (SUBSTRING(DenCl,1,1) =
                UPPER(SUBSTRING(DenCl,1,1))),
    CodFiscal CHAR(9) NOT NULL
        CONSTRAINT un_codfiscal UNIQUE
        CONSTRAINT ck_codfiscal CHECK (SUBSTRING(CodFiscal,1,1) =
                UPPER(SUBSTRING(CodFiscal,1,1))),
    Adresa VARCHAR(40)
        CONSTRAINT ck_adresa_clienti CHECK (SUBSTRING(Adresa,1,1) =
                UPPER(SUBSTRING(Adresa,1,1))),
    CodPost CHAR(6) NOT NULL
        CONSTRAINT fk_clienti_cp
                REFERENCES coduri_postale(CodPost) ON UPDATE CASCADE,
    Telefon VARCHAR(10)
    );

CREATE TABLE persoane (
    CNP CHAR(14)
        CONSTRAINT pk_persoane PRIMARY KEY
        CONSTRAINT ck_cnp CHECK (CNP = LTRIM(UPPER(CNP))),
    Nume VARCHAR(20) NOT NULL,
    Prenume VARCHAR(20) NOT NULL,
    Adresa VARCHAR(50)
        CONSTRAINT ck_adresa_persoane CHECK (SUBSTRING(Adresa,1,1) =
                UPPER(SUBSTRING(Adresa,1,1))),
    Sex CHAR(1) DEFAULT 'M' NOT NULL
        CONSTRAINT ck_sex CHECK (Sex IN ('F','M')),
    CodPost CHAR(6) NOT NULL CONSTRAINT fk_persoane_cp
        REFERENCES coduri_postale(CodPost) ON UPDATE CASCADE,
    TelAcasa VARCHAR(10),
    TelBirou VARCHAR(10),
    TelMobil VARCHAR(10),
    Email VARCHAR(30)
    );

CREATE TABLE persclienti (
```

```
    CNP CHAR(14) NOT NULL
        CONSTRAINT fk_persclienti_persoane REFERENCES persoane(CNP)
                ON UPDATE CASCADE,
    CodCl SMALLINT NOT NULL
        CONSTRAINT fk_persclienti_clienti REFERENCES clienti(CodCl)
    -- Aici ON UPDATE CASCADE nu functioneaza, adica genereaza eroare! --
    ,
    Functie VARCHAR(25) NOT NULL
        CONSTRAINT ck_functie CHECK (SUBSTRING(Functie,1,1) =
                UPPER(SUBSTRING(Functie,1,1))),
    CONSTRAINT pk_persclienti PRIMARY KEY (CNP, CodCl, Functie)
    );

CREATE TABLE produse (
    CodPr SMALLINT IDENTITY
        CONSTRAINT pk_produse PRIMARY KEY
        CONSTRAINT ck_codpr CHECK (CodPr > 0),
    DenPr VARCHAR(30) NOT NULL CONSTRAINT ck_denpr
        CHECK (SUBSTRING(DenPr,1,1) = UPPER(SUBSTRING(DenPr,1,1))),
    UM VARCHAR(10) NOT NULL,
    Grupa VARCHAR(15)
        CONSTRAINT ck_produse_grupa
                CHECK (SUBSTRING(Grupa,1,1) = UPPER(SUBSTRING(Grupa,1,1))),
    ProcTVA NUMERIC(2,2) DEFAULT .19 NOT NULL
    );

CREATE TABLE facturi (
    NrFact INTEGER
        CONSTRAINT pk_facturi PRIMARY KEY,
    DataFact SMALLDATETIME DEFAULT GETDATE() NOT NULL,
        CONSTRAINT ck_datafact CHECK (DataFact >= '2007-01-01'
                AND DataFact <= '2015-12-31'),
    CodCl SMALLINT NOT NULL
        CONSTRAINT fk_facturi_clienti
                REFERENCES clienti(CodCl) ON UPDATE CASCADE,
    Obs VARCHAR(50)
    );

CREATE TABLE liniifact (
     NrFact INTEGER
        CONSTRAINT fk_liniifact_facturi
                REFERENCES facturi(NrFact) ON UPDATE CASCADE,
    Linie TINYINT NOT NULL
        CONSTRAINT ck_linie CHECK (Linie > 0),
    CodPr SMALLINT NOT NULL
        CONSTRAINT fk_liniifact_produse
                REFERENCES produse(CodPr) ON UPDATE CASCADE,
    Cantitate NUMERIC(8) NOT NULL,
    PretUnit NUMERIC(9,2) NOT NULL,
    CONSTRAINT pk_liniifact PRIMARY KEY (NrFact,Linie)
    );

CREATE TABLE incasari (
    CodInc BIGINT IDENTITY (1234,1)
        CONSTRAINT pk_incasari PRIMARY KEY,
    DataInc SMALLDATETIME DEFAULT GETDATE() NOT NULL
        CONSTRAINT ck_datainc CHECK (DataInc >= '2007-01-01' AND
                DataInc <= '2015-12-31'),
    CodDoc CHAR(4) NOT NULL
```

```
        CONSTRAINT ck_coddoc CHECK(CodDoc = UPPER(LTRIM(CodDoc))),
    NrDoc VARCHAR(16) NOT NULL,
    DataDoc SMALLDATETIME DEFAULT GETDATE() - 7 NOT NULL
        CONSTRAINT ck_datadoc CHECK (DataDoc >= '2007-01-01'
                AND DataDoc <= '2015-12-31')
    );

CREATE TABLE incasfact (
    CodInc BIGINT
        CONSTRAINT fk_incasfact_incasari
                REFERENCES incasari(CodInc) ON UPDATE CASCADE,
    NrFact INTEGER
        CONSTRAINT fk_incasfact_facturi
                REFERENCES facturi(NrFact) ON UPDATE CASCADE,
    Transa NUMERIC(11,2) NOT NULL,
    CONSTRAINT pk_incasfact PRIMARY KEY (CodInc, NrFact)
    );

ALTER TABLE incasari ADD CONSTRAINT ck_date_incdoc CHECK (DataInc >= DataDoc);
```

Una dintre cele mai inoportune idei ale autorilor SQL Server este case sensitive-ul dus la extrem. Dacă numele unui atribut a fost scris, la CREATE TABLE cu litere mici, toate comenzile ALTER TABLE, INSERT, UPDATE, DELETE, SELECT următoare trebuie să facă referinţă la atributul respectiv exclusiv cu litere mici! Aşa că atenţie la modul de scriere a numelui atributelor din acest script! Dacă în ultima comandă - ALTER TABLE, în loc de *DataInc* am fi scris *datainc*, ne-am fi procopsit cu un mesaj de eroare.

3.4. Modificarea conţinutului

În acest moment, avem definită, în mare, structura bazei de date. Pasul următor, care se derulează pe toată durata vieţii (folosirii efective) unei BD este actualizarea conţinului. În BD TRIAJ trebuie să introducem efectiv datele despre medicii clinicii, pacienţii şi evenimentele de la camera de primire (triaj) a bolnavilor.

SQL prezintă comenzi dedicate modificării conţinutului unei tabele, înţelegând prin aceasta trei acţiuni prin care se actualizează baza:

a) adăugarea de noi linii la cele existente în tabelă;
b) ştergerea unor linii;
c) modificarea valorii unui atribut.

3.4.1. Adăugarea de linii în tabele

Comanda SQL de adăugare de noi linii este INSERT, cu un format destul de simplu. Iată câteva exemple:

```
INSERT INTO judete VALUES ('IS', 'Iasi', 'Moldova');
INSERT INTO coduri_postale VALUES ('700505', 'Iasi', 'IS');
INSERT INTO clienti VALUES (1001, 'Client 1 SRL', 'R1001', 'Tranzitiei, 13 bis',
        '700505', NULL);
```

```
INSERT INTO persoane VALUES ('CNP1', 'Ioan', 'Vasile',
        'I.L.Caragiale, 22', 'M', '700505', '123456', '987654', '0942222222', NULL);
INSERT INTO persclienti VALUES ('CNP1', 1001, 'Director general');
INSERT INTO produse VALUES (1, 'Produs 1','buc', 'Tigari', .19);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1111, DATE'2008-01-01',
        1001);
INSERT INTO facturi (NrFact, DataFact, CodCl)
        VALUES (1121, DATE'2008-08-07', 1001);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (1111, 1, 1, 50, 1000);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (1111, 2, 2, 75, 1050);
INSERT INTO incasari VALUES (1234, DATE'2008-08-15', 'OP', '111', DATE'2008-10-08');
INSERT INTO incasfact VALUES (1234, 1111, 53996);
```

Ordinea valorilor din clauza VALUES trebuie să fie identică cu cea declarată la crearea (sau modificarea structurii) tabelelor. Modificarea este posibilă numai prin enumerarea, după numele tabelei, a atributelor ce vor primi valorile specificate. Pentru tabelele FACTURI şi LINIIFACT au fost incluse în clauza VALUES mai puţine valori decât atributele tabelei. Este obligatorie, în aceste cazuri, precizarea atributelor care vor primi valorile. Restul, adică cele nespecificate, vor avea, pe liniile respective, valori NULL.

Dacă valorile specificate în clauza VALUES vin în contradicţie cu măcar o restricţie declarată, se afişează un mesaj de eroare, iar comanda INSERT este respinsă. Spre exemplu, *INSERT INTO judete VALUES ('IS', 'Iasi', 'Mldova')* încalcă restricţia definită pentru atributul Regiune (CK_REGIUNE), iar rezultatul din DB2 este prezentat în figura 3.6.

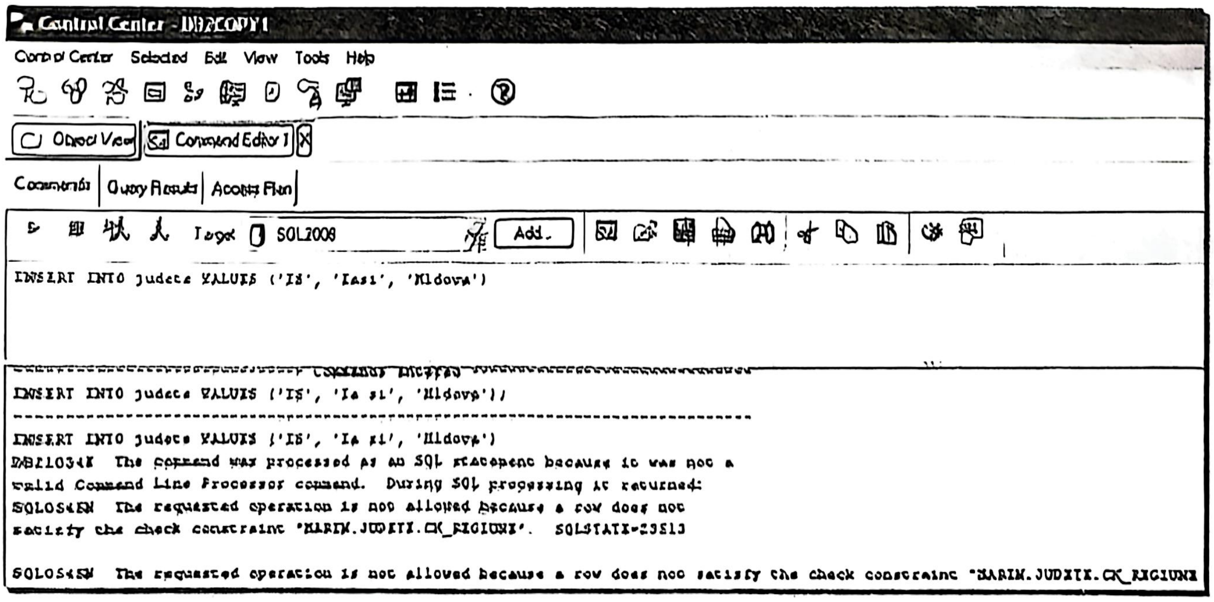


Figura 3.6. Violarea restricţiei CK_REGIUNE (în DB2)

3.4.2. Ştergerea liniilor

Comanda SQL pentru ştergerea uneia sau mai multor linii dintr-o tabelă este DELETE, al cărei format general este: *DELETE [FROM] nume-tabelă [WHERE predicat]*. Din tabelă vor fi şterse toate liniile care îndeplinesc condiţia specificată în predicatul din clauza WHERE. Astfel, *DELETE FROM facturi WHERE NrFact = 1122* elimină din tabela FACTURI toate înregistrările în care valoarea atributului NrFact este egală cu 1122, altfel spus, din tabela FACTURI este ştearsă linia care se referă la factura 1122. Când pentru această factură există înregistrări-copil în LINIIFACT sau INCASFACT, SGBD-ul ţine cont de opţiunea declarată la crearea tabelei. Dacă s-a specificat (implicit sau explicit) ON DELETE RESTRICT, ştergerea este interzisă. În cazul ON DELETE CASCADE sunt şterse, odată cu linia din FACTURI, toate liniile-copil din LINIIFACT şi INCASFACT.

Comanda *DELETE FROM judete WHERE Regiune = 'Moldova'* elimină din baza de date toate judeţele din Moldova (şterge toate liniile tabelei JUDEŢE în care valoarea atributului Regiune este Moldova).

3.4.3. Modificarea valorilor unor atribute

Pentru a modifica valoarea unuia sau mai multor atribute pe una sau mai multe linii dintr-o tabelă se foloseşte comanda UPDATE cu formatul general (simplificat): *UPDATE tabelă SET atribut1 = expresie1 [, atribut2= expresie2 ...,] WHERE predicat.* Modificarea se va produce pe toate liniile tabelei care îndeplinesc condiţia formulată prin predicat. Exemple:

- noul număr de telefon al clientului ce are codul 1001 este 0232-313131:

```
UPDATE CLIENTI SET Telefon = '0232-313131' WHERE CodCl = 1001
```

- în cadrul unei noi relaxări fiscale, se decide creşterea procentului TVA de la 19% la 22% pentru toate produsele. Atributul modificat este ProcTVA din tabela PRODUSE, pe toate liniile:

```
UPDATE PRODUSE SET ProcTVA = .22
```

3.4.4. Scripturile de populare în DB2

În practică, popularea cu înregistrări se face interactiv, folosind meniuri şi formulare prin care utilizatorii (de obicei, total nepricepuţi în ale bazelor de date) preiau informaţiile. Noi însă, neştiind să construim formulare, vom adăuga/modifica/şterge „manual" înregistrări în/din tabele, folosind cele trei comenzi pe care le-am mai pomenit, INSERT, UPDATE şi DELETE.

Spuneam, la crearea tabelelor, că atributele cu valori autogenerate au un regim special la editare. Astfel, în tabela DOCTORI, IdDoctor este un asemenea atribut. Dacă am urma logica INSERT-ului, am fi tentaţi să scriem:

```
INSERT INTO doctori VALUES (1, 'Vasilcu Ionel', 'chirurgie', '1965-11-11');
```

Problema vine de la faptul că, la crearea tabelei, am declarat IdDoctor ca având valori generate, în timp ce acum, vrem să atribuim acestui atribut valoarea 1 prin

INSERT. Chiar dacă valoarea declarată în comanda INSERT coincide cu valoarea generată automat (adică 1), serverul se va împotrivi – vezi figura 3.7.

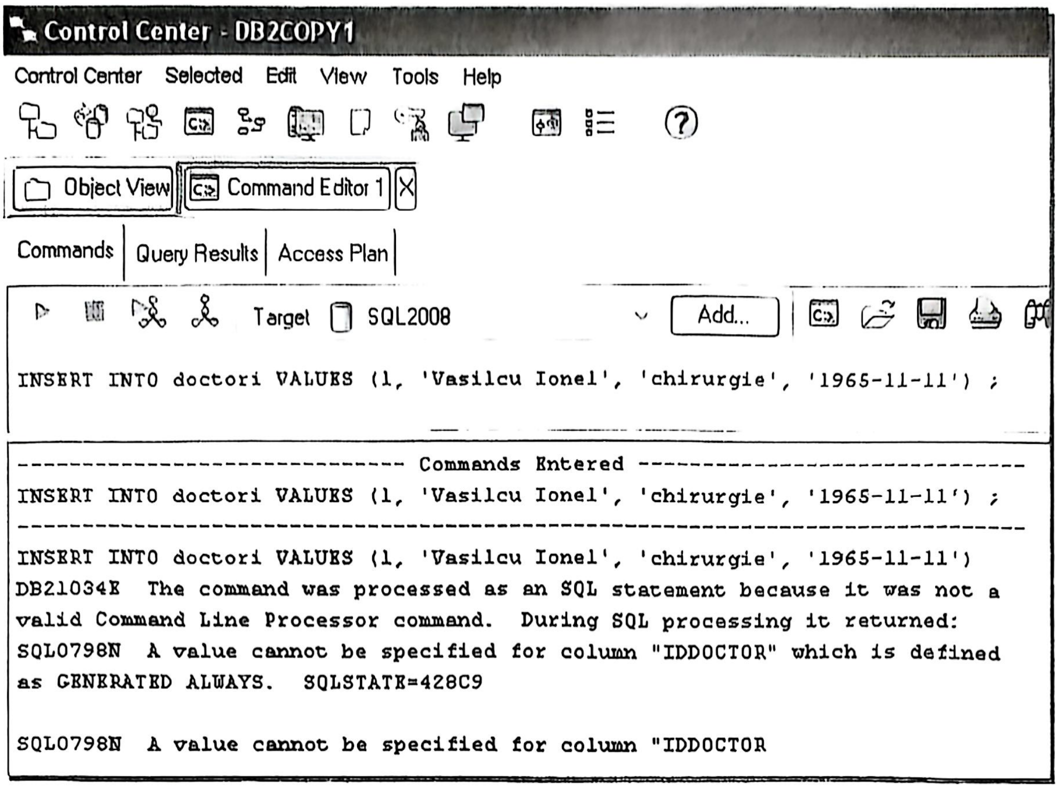


Figura 3.7. Eroare cauzată de încercarea de stabilire prin INSERT a valorii unui atribut cu valori autogenerate (DB2)

Reacţia serverului este îndreptăţită. Listingul 3.9 conţine scriptul de populare a celor patru tabele din BD TRIAJ. Comenzile DELETE de început asigură „curăţarea" tabelelor, în caz că aveau vreo înregistrare dinainte. În lipsa acestora, la următoarea lansare a scriptului, s-ar fi violat restricţia de cheie primară pentru tabela GĂRZI. Desprea comanda de final, COMMIT, vom discuta peste câteva paragrafe.

Listing 3.9. Script DB2 de populare a tabelelor BD TRIAJ

```
DELETE FROM triaj;
DELETE FROM pacienti;
DELETE FROM garzi;
DELETE FROM doctori;

INSERT INTO doctori (numedoctor, specialitate, datanasterii)
    VALUES ('Vasilcu Ionel', 'chirurgie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate, datanasterii)
    VALUES ('Georgescu Mircea', 'hepatologie', '1966-01-12');
INSERT INTO doctori (numedoctor, specialitate, datanasterii)
    VALUES ('Zahir Tudorel', 'chirurgie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate, datanasterii) VALUES
    ('Bostan Vasile', 'cardiologie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate, datanasterii)
    VALUES ('Popescu Ionela', 'boli interne', '1965-11-11');

INSERT INTO garzi VALUES (2, '2008-01-03 7:00:00', '2008-01-03 19:00:00');
```

```
INSERT INTO garzi VALUES (3, '2008-01-03 19:00:00', '2008-01-04 7:00:00');
INSERT INTO garzi VALUES (4, '2008-01-04 07:00:00', '2008-01-04 18:00:00');
INSERT INTO garzi VALUES (5, '2008-01-04 18:00:00', '2008-01-05 06:00:00');
INSERT INTO garzi VALUES (2, '2008-01-05 6:00:00', '2008-01-05 14:00:00');
INSERT INTO garzi VALUES (3, '2008-01-05 14:00:00', '2008-01-05 22:00:00');
INSERT INTO garzi VALUES (5, '2008-01-05 22:00:00', '2008-01-06 9:00:00');

INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Stroe Mihaela', 2601228390834, 'Bd. Cantemir, 32, Bl.G4, Sc.C, Ap.4', 'MR366766');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Buzatu Corneliu', 1650512370514, 'Str. Desprimaveririi, 112', 'MX456783');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
     ('Spineanu Marius', 5010101380625, 'Bd. Stefan cel Mare, 4, Bl.I1, Sc.A, Ap.24', 'MX213345');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Bagdasar Adela', 2601002250611, 'Str. Primaverii, 17', 'MX345678');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Ionascu Ionela', 6001122390199, 'Str. Florilor', 'MX654322');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Popescu Maria-Mirabela', 2721231300888,
        'Bd. 22 Decembrie, 2, Bl.5, Sc.B, Ap.21', 'MX765432');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Stroescu Mihaela Oana', 6020719120545, 'Str. Lenei Nr. 234', 'MX876567');
INSERT INTO pacienti (numepacient, cnp, adresa, serie_nr_act_identitate) VALUES
    ('Cazan Ana Maria', 2690202200345, 'Bd. Independentei, 87, Bl.K3, Sc.A, Ap.34', 'MX987789');

INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 7:18:00', 1, 'dureri de stomac intense', NULL, 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 8:45:00', 2, 'febra puternica, varsaturi', 'penicilina', 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 12:45:00', 3, 'deranjament stomacal', 'scobutil', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 20:45:00', 4, 'palpitatii cardiace', 'linistin', 'cardiologie');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 1:28:00', 5, 'plaga profunda picior drept', 'antibiotice, pansament', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 10:45:00', 3, 'contractii stomacale, varsaturi', NULL, 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 11:20:00', 7, 'fata de culoare galbena, ameteli', NULL, 'hepatologie');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 22:45:00', 8, 'dureri articulare', 'scobutil', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-05 6:18:00', 5, 'febra puternica, delir', 'penicilina', 'chirurgie');
COMMIT;
```

Ei bine, chiar dacă scriptul începe cu ştergerea eventualelor linii din cele patru tabele, la a doua (a treia, ...) lansare în execuţie tot nu scăpăm de probleme - vezi figura 3.8.

Această eroare dezvăluie cea mai importantă durere de cap creată de atributele autogenerate, şi anume că nu le putem controla. Astfel, la a doua lansare, chiar dacă toate liniile au fost şterse, valorile de la 1 la 5 pentru IdDoctor au fost consumate, aşa că, acum, id-urile celor cinci medici sunt de la 6 la 10. Baiul derivă de la faptul că IdDoctor este cheie străină în tabela GĂRZI. Or, şi la a doua lansare a scriptului, prima comandă de inserare în GĂRZI rămâne tot *INSERT INTO garzi*

VALUES (2, '2008-01-03 7:00:00', '2008-01-03 19:00:00'), iar valoarea 2 nu se mai regăseşte în tabela-părinte (DOCTORI).

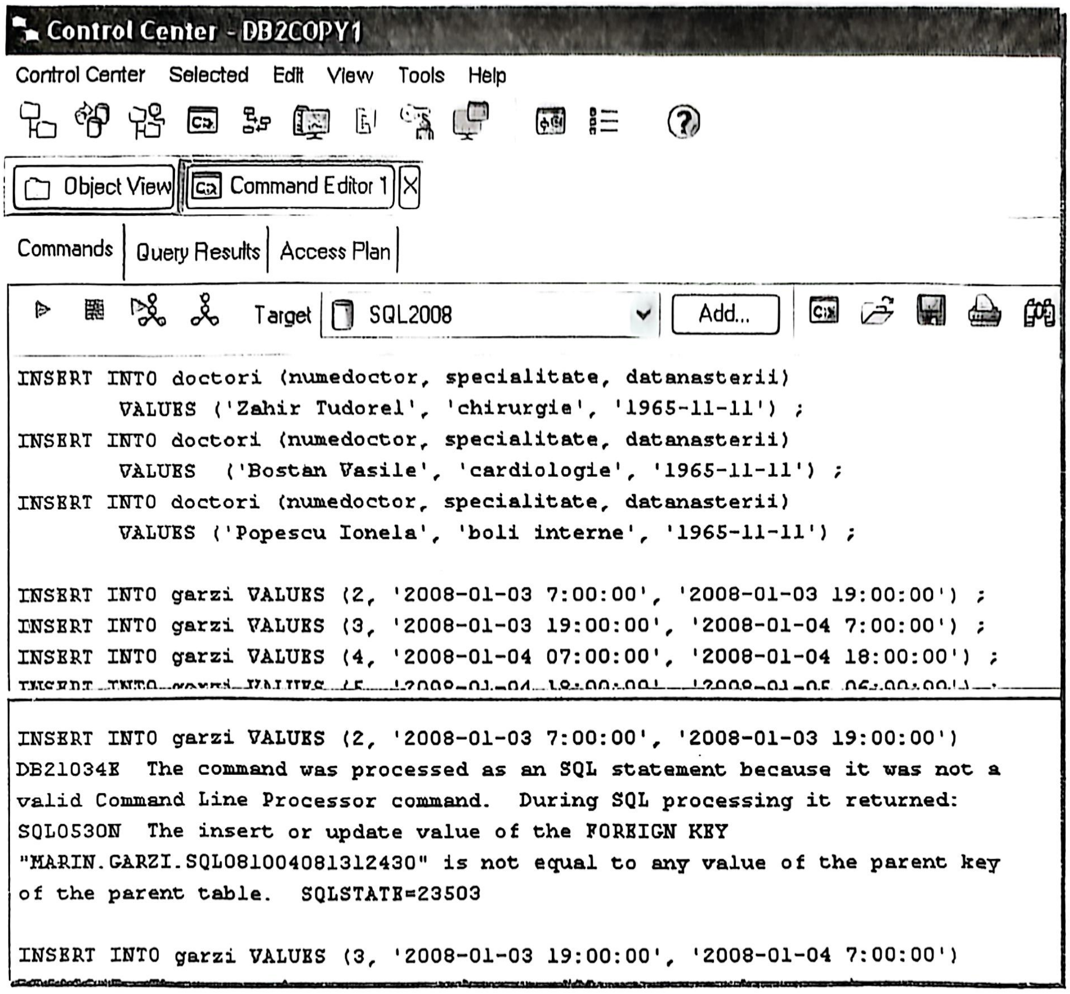


Figura 3.8. Necazuri cu atributele ale căror valori sunt generate automat (DB2)

Cum rezolvăm această problemă enervantă? Soluţia cea mai elegantă este ca, la inserarea în tabelele-copil, valorile cheilor străine care reprezintă (în tabela--părinte) atribute cu valori generate automat, să nu fie specificate prin constante, ci prin subsconsultări:

```
INSERT INTO garzi VALUES ( (SELECT IdDoctor FROM doctori WHERE
        numedoctor='Vasilcu Ionel'), '2008-01-03 7:00:00', '2008-01-03 19:00:00');
```

Din păcate, nivelul nostru de SQL este, în acest paragraf, mult sub cerinţele acestei sintaxe, aşa încât nu ne rămâne decât soluţia ruşinoasă, adică relansarea scriptului de creare (listing 3.1), şi apoi a celui de populare (listing 3.9) doar o singură dată. Ar mai fi însă şi o a treia soluţie, pe care o folosim în scriptul de populare a tabelelor bazei de date VÂNZĂRI - listing 3.10. DB2 pune la dispoziţie funcţia IDENTITY_VAL_LOCAL care „ţine minte" ultima valoare a unei coloane autogenerate. În aceste condiţii, după fiecare înregistrare-părinte în care cheia primară este autogenerată, vom insera toate înregistrările-copil ale acesteia, ceea ce este destul de neplăcut. Iar dacă am fi avut în bază o tabelă cu două chei străine, autogenerate în tabelele-părinte, chiar că lucrurile luau o întorsătură interesantă.

O premieră în listingul 3.10 ține de posibilitatea oferită de DB2 ca **într-o clauză** VALUES a comenzii INSERT să fie specificate, simultan, **mai multe linii. Astfel,** cele şase judeţe sunt introduse în tabelă cu o singură comandă. De fapt, **majoritatea** liniilor introduse în tabele folosesc această facilitate.

Listing 3.10. Script DB2 de populare a tabelelor BD VÂNZĂRI

```
DELETE FROM incasfact;
DELETE FROM incasari;
DELETE FROM liniifact;
DELETE FROM facturi;
DELETE FROM produse;
DELETE FROM persclienti;
DELETE FROM persoane;
DELETE FROM clienti;
DELETE FROM coduri_postale;
DELETE FROM judete;

INSERT INTO judete VALUES ('IS', 'Iasi', 'Moldova'), ('VN', 'Vrancea', 'Moldova'),
    ('NT', 'Neamt', 'Moldova'), ('SV', 'Suceava', 'Moldova'),
    ('VS', 'Vaslui', 'Moldova'), ('TM', 'Timis', 'Banat');

INSERT INTO coduri_postale VALUES ('700505', 'Iasi', 'IS'), ('700510', 'Iasi', 'IS'),
    ('700515', 'Iasi', 'IS'), ('701150', 'Pascani', 'IS'), ('706500', 'Vaslui', 'VS'),
    ('706510', 'Vaslui', 'VS'), ('705300', 'Focsani', 'VN'), ('705310', 'Focsani', 'VN'),
    ('706400', 'Birlad', 'VS'), ('705800', 'Suceava', 'SV'), ('705550', 'Roman', 'NT'),
    ('701900', 'Timisoara', 'TM');

INSERT INTO persoane VALUES
    ('CNP1', 'Ioan', 'Vasile', 'I.L.Caragiale, 22', 'M', '700505', 123456', '987654', '094222222', NULL),
    ('CNP2', 'Vasile', 'Ion', NULL, 'M', '700505', '234567', '876543', '094222223', 'Ion@a.ro'),
    ('CNP3', 'Popovici', 'Ioana', 'V.Micle, Bl.I, Sc.B,Ap.2', 'F', '701150', '345678', NULL, '094222224',
        NULL);
INSERT INTO persoane VALUES
    ('CNP4', 'Lazar', 'Caraion', 'M.Eminescu, 42', 'M', '706500', '456789', NULL, '094222225', NULL),
    ('CNP5', 'Iurea', 'Simion', 'I.Creanga, 44 bis', 'M', '706500', '567890', '543210', NULL, NULL),
    ('CNP6', 'Vasc', 'Simona', 'M.Eminescu, 13', 'F', '701150', NULL, '432109', '094222227', NULL),
    ('CNP7', 'Popa', 'Ioanid', 'I.Ion, Bl.H2, Sc.C, Ap.45', 'M', '701900', '789012', '321098', NULL, NULL),
    ('CNP8', 'Bogacs', 'Ildiko', 'I.V.Viteazu, 67', 'F', '705550', '890123', '210987', '094222229', NULL),
        ('CNP9', 'Ioan', 'Vasilica', 'Garii, Bl.B4, Sc.A, Ap.1', 'F','701900', '901234', '109876',
'094222230',
            NULL);

-- primul client:
INSERT INTO clienti VALUES (DEFAULT, 'Client 1 SRL', 'R1001', 'Tranzitiei, 13 bis',
    '700505', NULL);
-- liniile-copil din tabelele PERSCLIENTI si FACTURI:
INSERT INTO persclienti VALUES ('CNP1', identity_val_local(), 'Director general');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES
    (1111, '2007-08-01', identity_val_local()), (1115,'2007-08-02', identity_val_local()),
    (1117, '2007-08-03', identity_val_local()), (1118, '2007-08-04', identity_val_local()),
    (1120, '2007-08-07', identity_val_local()), (2111, '2007-08-14', identity_val_local()),
    (2115, '2007-08-15', identity_val_local()), (2117, '2007-08-16', identity_val_local()),
    (2118, '2007-08-16', identity_val_local()), (3111, '2007-09-01', identity_val_local()),
    (3115, '2007-09-02', identity_val_local()), (3117, '2007-09-10', identity_val_local()),
    (3118, '2007-09-17', identity_val_local());

-- al doilea client:
```

```
INSERT INTO clienti (dencl, codfiscal, codpost, telefon)
    VALUES ('Client 2 SA', 'R1002', '700505', '0232212121');
-- inregistrarile-copil al celui de-al doilea client:
INSERT INTO persclienti VALUES ('CNP2', identity_val_local(), 'Director general'),
    ('CNP3', identity_val_local(), 'Sef aprovizionare');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES
(1113, '2007-08-01', identity_val_local()), (2113, '2007-08-14', identity_val_local()),
    (3113, '2007-09-02', identity_val_local());

-- al treilea client:
INSERT INTO clienti VALUES
    (DEFAULT, 'Client 3 SRL', 'R1003', 'Prosperitatii, 22', '706500','0235222222');
-- inregistrarile-copil al celui de-al treilea client:
INSERT INTO persclienti VALUES ('CNP4', identity_val_local(), 'Sef aprovizionare'),
    ('CNP5', identity_val_local(), 'Director financiar');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES
(1119, '2007-08-07', identity_val_local()), (2119, '2007-08-21', identity_val_local()),
 (3119, '2007-10-07', identity_val_local());

-- al patrulea client:
INSERT INTO clienti (dencl, codfiscal, adresa, codpost) VALUES
    ('Client 4', 'R1004', 'Sapientei, 56', '701150');
-- inregistrarile-copil al celui de-al patrulea client:
INSERT INTO persclienti VALUES ('CNP6', identity_val_local(), 'Director general');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES
    (1121, '2007-08-07', identity_val_local()), (2121, '2007-08-21', identity_val_local());

-- al cincilea client:
INSERT INTO clienti VALUES (DEFAULT, 'Client 5 SRL', 'R1005', NULL, '701900', '0256111111');
-- inregistrarile-copil al celui de-al cincilea client:
INSERT INTO persclienti VALUES ('CNP7', identity_val_local(), 'Sef aprovizionare');
INSERT INTO facturi (nrfact, datafact, codcl, obs) VALUES
    (1112, '2007-08-01', identity_val_local(), 'Probleme cu transportul'),
    (1122, '2007-08-07', identity_val_local(), NULL),
    (2112, '2007-08-14', identity_val_local(),'Probleme cu transportul'),
    (2122, '2007-08-22', identity_val_local(), NULL),
    (3112, '2007-09-01', identity_val_local(), 'Probleme cu transportul');

-- al saselea client:
INSERT INTO clienti VALUES (DEFAULT, 'Client 6 SA', 'R1006', 'Pacientei, 33', '705550', NULL);
-- inregistrarile-copil al celui de-al saselea client:
INSERT INTO persclienti VALUES ('CNP8', identity_val_local(), 'Director financiar');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES
    (1114, '2007-08-01', identity_val_local());

-- al saptelea client:
INSERT INTO clienti VALUES
    (DEFAULT, 'Client 7 SRL', 'R1007', 'Victoria Capitalismului, 2', '701900', '0256121212');
-- inregistrarile-copil al celui de-al saptelea client:
INSERT INTO persclienti VALUES ('CNP9', identity_val_local(), 'Sef aprovizionare');
INSERT INTO facturi (nrfact, datafact, codcl, obs) VALUES
    (1116, '2007-08-02', identity_val_local(), 'Pretul propus initial a fost modificat'),
    (2116, '2007-08-15', identity_val_local(), 'Pretul propus initial a fost modificat'),
    (3116, '2007-09-10', identity_val_local(), 'Pretul propus initial a fost modificat');

-- primul produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 1','buc', 'Tigari', .19);
-- liniile in care apare primul produs:
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit) VALUES
```

```
    (1111, 1, identity_val_local(), 50, 1000), (1117, 2, identity_val_local(), 100, 950),
    (1118, 2, identity_val_local(), 150, 930), (2111, 1, identity_val_local(), 57, 1000),
    (2117, 2, identity_val_local(), 110, 950), (2118, 2, identity_val_local(), 120, 930),
    (3111, 1, identity_val_local(), 57, 1000), (3117, 2, identity_val_local(), 110, 950),
    (3118, 2, identity_val_local(), 120, 930);

-- al doilea produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 2','kg', 'Bere', 0.09);
-- liniile in care apare al doilea produs:
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit) VALUES
    (1111, 2, identity_val_local(), 75, 1050), (1112, 1, identity_val_local(), 80, 1030),
    (1113, 1, identity_val_local(), 100, 975), (1114, 1, identity_val_local(), 70, 1070),
    (1115, 1, identity_val_local(), 150, 925), (1116, 1, identity_val_local(), 125, 930),
    (1117, 1, identity_val_local(), 100, 1000), (1118, 1, identity_val_local(), 30, 1100),
    (1119, 1, identity_val_local(), 35, 1090), (1120, 1, identity_val_local(), 80, 1120),
    (1121, 2, identity_val_local(), 100, 1050), (2111, 2, identity_val_local(), 79, 1050),
    (2112, 1, identity_val_local(), 85, 1030), (2113, 1, identity_val_local(), 120, 975),
    (2115, 1, identity_val_local(), 110, 925), (2116, 1, identity_val_local(), 135, 930),
    (2117, 1, identity_val_local(), 150, 1000), (2118, 1, identity_val_local(), 39, 1100),
    (2119, 1, identity_val_local(), 35, 1090), (2121, 2, identity_val_local(), 103, 1050),
    (3111, 2, identity_val_local(), 79, 1050), (3112, 1, identity_val_local(), 85, 1030),
    (3113, 1, identity_val_local(), 120, 975), (3115, 1, identity_val_local(), 110, 925),
    (3116, 1, identity_val_local(), 135, 930), (3117, 1, identity_val_local(), 150, 1000),
    (3118, 1, identity_val_local(), 39, 1100), (3119, 1, identity_val_local(), 35, 1090);

-- al treilea produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 3','kg', 'Bere', 0.19);
-- liniile in care apare al treilea produs:
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit) VALUES
    (1112, 2, identity_val_local(), 40, 750), (1119, 2, identity_val_local(), 40, 700),
    (2112, 2, identity_val_local(), 65, 750), (2119, 2, identity_val_local(), 40, 700),
    (3112, 2, identity_val_local(), 65, 750), (3119, 2, identity_val_local(), 40, 700);

-- al patrulea produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 4','l', 'Dulciuri', .09);
-- liniile in care apare al patrulea produs:
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit) VALUES
    (1114, 2, identity_val_local(), 30, 1705), (1119, 3, identity_val_local(), 50, 1410),
    (2119, 3, identity_val_local(), 55, 1410), (3119, 3, identity_val_local(), 55, 1410);

-- al cincilea produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 5','buc', 'Tigari', .19);
-- liniile in care apare al cincilea produs:
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit) VALUES
    (1111, 3, identity_val_local(), 500, 7060), (1114, 3, identity_val_local(), 700, 7064),
    (1119, 4, identity_val_local(), 750, 6300), (1121, 1, identity_val_local(), 550, 7064),
    (2111, 3, identity_val_local(), 510, 7060), (2119, 4, identity_val_local(), 755, 6300),
    (2121, 1, identity_val_local(), 550, 7064), (3111, 3, identity_val_local(), 510, 7060),
    (3119, 4, identity_val_local(), 755, 6300);

-- al saselea produs:
INSERT INTO produse VALUES (DEFAULT, 'Produs 6','p250g', 'Cafea', .19);
-- nu e nicio linie in care apare al saselea produs

-- prima incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-15', 'OP', '111', '2007-08-10' );
-- facturile pentru prima incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1111, 53996), (identity_val_local(), 1118,
101975);
```

```
-- a doua incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-15', 'CHIT', '222', '2007-08-15');
-- facturile pentru a doua incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1112, 125516);

-- a treia incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-16', 'OP', '333', '2007-08-09');
-- facturile pentru a treia incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1117, 9754),
    (identity_val_local(), 1118, 100000), (identity_val_local(), 1120, 7315);

-- a patra incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-17', 'CEC', '444', '2007-08-10');
-- facturile pentru a patra incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1117, 9754);

-- a cincea incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-17', 'OP', '555', '2007-08-10');
-- facturile pentru a cincea incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1113, 106275);

-- a sasea incasare:
INSERT INTO incasari VALUES (DEFAULT, '2007-08-18', 'OP', '666', '2007-08-11');
-- facturile pentru a sasea incasare:
INSERT INTO incasfact VALUES (identity_val_local(), 1117, 3696);

COMMIT;
```

3.4.5. Scripturile de populare în Oracle

În Oracle, inserările (listingurile 3.11 şi 3.12) sunt mult mai relaxate, graţie inexistenţei atributelor cu valori autogenerate. Constantele de tip dată calendaristică sunt prefixate, după cum am văzut, de cuvântul DATE, iar cele de tip moment de TIMESTAMP. Beneficiind de faptul că putem scrie mai multe comenzi pe aceeaşi linie, încercăm să mai scurtăm cât de cât lungimea scripturilor, chiar dacă lizibilitatea se reduce considerabil.

Listing 3.11. Script Oracle de populare a tabelelor BD TRIAJ

```
DELETE FROM triaj; DELETE FROM pacienti; DELETE FROM  garzi;
DELETE FROM  doctori;

INSERT INTO doctori VALUES (1, 'Vasilcu Ionel', 'chirurgie', DATE'1965-11-11');
INSERT INTO doctori VALUES (2, 'Georgescu Mircea', 'hepatologie', DATE'1966-01-12');
INSERT INTO doctori VALUES (3, 'Zahir Tudorel', 'chirurgie', DATE'1965-11-11');
INSERT INTO doctori VALUES (4, 'Bostan Vasile', 'cardiologie', DATE'1965-11-11');
INSERT INTO doctori VALUES (5, 'Popescu Ionela', 'boli interne', DATE'1965-11-11');

INSERT INTO garzi VALUES
    (1, TIMESTAMP'2008-01-03 7:00:00', TIMESTAMP'2008-01-03 19:00:00');
INSERT INTO garzi VALUES
    (2, TIMESTAMP'2008-01-03 19:00:00', TIMESTAMP'2008-01-04 7:00:00');
INSERT INTO garzi VALUES
    (3, TIMESTAMP'2008-01-04 07:00:00', TIMESTAMP'2008-01-04 18:00:00');
INSERT INTO garzi VALUES
```

```
    (4, TIMESTAMP'2008-01-04 18:00:00', TIMESTAMP'2008-01-05 06:00:00');
INSERT INTO garzi VALUES
    (1, TIMESTAMP'2008-01-05 6:00:00', TIMESTAMP'2008-01-05 14:00:00');
INSERT INTO garzi VALUES
    (2, TIMESTAMP'2008-01-05 14:00:00', TIMESTAMP'2008-01-05 22:00:00');
INSERT INTO garzi VALUES
    (5, TIMESTAMP'2008-01-05 22:00:00', TIMESTAMP'2008-01-06 9:00:00');

INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (1, 'Stroe Mihaela', 2601228390834, 'Bd. Cantemir, 32, Bl.G4, Sc.C, Ap.4', 'MR366766');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (2, 'Buzatu Corneliu', 1650512370514, 'Str. Desprimaveririi, 112', 'MX456783');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (3, 'Spineanu Marius', 5010101380625, 'Bd. Stefan cel Mare, 4, Bl.I1, Sc.A, Ap.24',
        'MX213345');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (4, 'Bagdasar Adela', 2601002250611, 'Str. Primaverii, 17', 'MX345678');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (5, 'Ionascu Ionela', 6001122390199, 'Str. Florilor', 'MX654322');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (6, 'Popescu Maria-Mirabela', 2721231300888, 'Bd. 22 Decembrie, 2, Bl.5, Sc.B, Ap.21',
        'MX765432');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (7, 'Stroescu Mihaela Oana', 6020719120545, 'Str. Lenei Nr. 234', 'MX876567');
INSERT INTO pacienti (idpacient,numepacient, cnp, adresa, serie_nr_act_identitate)
    VALUES (8, 'Cazan Ana Maria', 2690202200345, 'Bd. Independentei, 87, Bl.K3, Sc.A, Ap.34',
        'MX987789');

INSERT INTO triaj VALUES (1, TIMESTAMP'2008-01-03 7:18:00', 1,
    'dureri de stomac intense', NULL, 'boli interne');
INSERT INTO triaj VALUES (2, TIMESTAMP'2008-01-03 8:45:00', 2,
    'febra puternica, varsaturi', 'penicilina', 'boli interne');
INSERT INTO triaj VALUES (3, TIMESTAMP'2008-01-03 12:45:00', 3,
    'deranjament stomacal', 'scobutil', NULL);
INSERT INTO triaj VALUES (4, TIMESTAMP'2008-01-03 20:45:00', 4,
    'palpitatii cardiace', 'linistin', 'cardiologie');
INSERT INTO triaj VALUES (5, TIMESTAMP'2008-01-04 1:28:00', 5,
    'plaga profunda picior drept', 'antibiotice, pansament', NULL);
INSERT INTO triaj VALUES (6, TIMESTAMP'2008-01-04 10:45:00', 3,
    'contractii stomacale, varsaturi', NULL, 'boli interne');
INSERT INTO triaj VALUES (7, TIMESTAMP'2008-01-04 11:20:00', 7,
    'fata de culoare galbena, ameteli', NULL, 'hepatologie');
INSERT INTO triaj VALUES (8, TIMESTAMP'2008-01-04 22:45:00', 8,
    'dureri articulare', 'scobutil', NULL);
INSERT INTO triaj VALUES (9, TIMESTAMP'2008-01-05 6:18:00', 5,
    'febra puternica, delir', 'penicilina', 'chirurgie');

COMMIT;
```

Din păcate, în Oracle nu putem enumera mai multe linii în clauza VALUES a comenzii INSERT, aşa încât listingul 3.12 are o lungime apreciabilă.

Listing 3.12. Script Oracle de populare a tabelelor BD VÂNZĂRI

```
DELETE FROM incasfact; DELETE FROM incasari; DELETE FROM liniifact;
DELETE FROM facturi; DELETE FROM produse; DELETE FROM persclienti;
DELETE FROM persoane; DELETE FROM clienti; DELETE FROM coduri_postale;
DELETE FROM judete;
```

```
INSERT INTO judete VALUES ('IS', 'Iasi', 'Moldova');
INSERT INTO judete VALUES ('VN', 'Vrancea', 'Moldova');
INSERT INTO judete VALUES ('NT', 'Neamt', 'Moldova');
INSERT INTO judete VALUES ('SV', 'Suceava', 'Moldova');
INSERT INTO judete VALUES ('VS', 'Vaslui', 'Moldova');
INSERT INTO judete VALUES ('TM', 'Timis', 'Banat');

INSERT INTO coduri_postale VALUES ('700505', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('700510', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('700515', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('701150', 'Pascani', 'IS');
INSERT INTO coduri_postale VALUES ('706500', 'Vaslui', 'VS');
INSERT INTO coduri_postale VALUES ('706510', 'Vaslui', 'VS');
INSERT INTO coduri_postale VALUES ('705300', 'Focsani', 'VN');
INSERT INTO coduri_postale VALUES ('705310', 'Focsani', 'VN');
INSERT INTO coduri_postale VALUES ('706400', 'Barlad', 'VS');
INSERT INTO coduri_postale VALUES ('705800', 'Suceava', 'SV');
INSERT INTO coduri_postale VALUES ('705550', 'Roman', 'NT');
INSERT INTO coduri_postale VALUES ('701900', 'Timisoara', 'TM');

INSERT INTO clienti VALUES (1001, 'Client 1 SRL', 'R1001', 'Tranzitiei, 13 bis',
    '700505', NULL);
INSERT INTO clienti (codcl, dencl, codfiscal, codpost, telefon)
    VALUES (1002,'Client 2 SA', 'R1002', '700505', '0232212121');
INSERT INTO clienti VALUES (1003, 'Client 3 SRL', 'R1003', 'Prosperitatii, 22',
    '706500','0235222222');
INSERT INTO clienti (codcl, dencl, adresa, codfiscal, codpost)
    VALUES (1004, 'Client 4', 'Sapientei, 56', 'R1004', '701150');
INSERT INTO clienti VALUES (1005, 'Client 5 SRL', 'R1005', NULL,
    '701900', '0256111111');
INSERT INTO clienti VALUES (1006, 'Client 6 SA', 'R1006', 'Pacientei, 33',
    '705550', NULL);
INSERT INTO clienti VALUES (1007, 'Client 7 SRL', 'R1007', 'Victoria Capitalismului, 2',
    '701900', '0256121212');

INSERT INTO persoane VALUES ('CNP1', 'Ioan', 'Vasile', 'I.L.Caragiale, 22', 'M',
    '700505', '123456', '987654', '094222222', NULL);
INSERT INTO persoane VALUES ('CNP2', 'Vasile', 'Ion', NULL, 'M',
    '700505', '234567', '876543', '094222223', 'Ion@a.ro');
INSERT INTO persoane VALUES ('CNP3', 'Popovici', 'Ioana', 'V.Micle, Bl.I, Sc.B,Ap.2', 'F',
    '701150', '345678', NULL, '094222224', NULL);
INSERT INTO persoane VALUES ('CNP4', 'Lazar', 'Caraion', 'M.Eminescu, 42', 'M',
    '706500', '456789', NULL, '094222225', NULL);
INSERT INTO persoane VALUES ('CNP5', 'Iurea', 'Simion', 'I.Creanga, 44 bis', 'M',
    '706500', '567890', '543210', NULL, NULL);
INSERT INTO persoane VALUES ('CNP6', 'Vasc', 'Simona', 'M.Eminescu, 13', 'F',
    '701150', NULL, '432109', '094222227', NULL);
INSERT INTO persoane VALUES ('CNP7', 'Popa', 'Ioanid', 'I.Ion, Bl.H2, Sc.C, Ap.45', 'M',
    '701900', '789012', '321098', NULL, NULL);
INSERT INTO persoane VALUES ('CNP8', 'Bogacs', 'Ildiko', 'I.V.Viteazu, 67', 'F',
    '705550', '890123', '210987', '094222229', NULL);
INSERT INTO persoane VALUES ('CNP9', 'Ioan', 'Vasilica', 'Garii, Bl.B4, Sc.A, Ap.1', 'F',
    '701900', '901234', '109876', '094222230', NULL);

INSERT INTO persclienti VALUES ('CNP1', 1001, 'Director general');
INSERT INTO persclienti VALUES ('CNP2', 1002, 'Director general');
INSERT INTO persclienti VALUES ('CNP3', 1002, 'Sef aprovizionare');
INSERT INTO persclienti VALUES ('CNP4', 1003, 'Sef aprovizionare');
```

```
INSERT INTO persclienti VALUES ('CNP5', 1003, 'Director financiar');
INSERT INTO persclienti VALUES ('CNP6', 1004, 'Director general');
INSERT INTO persclienti VALUES ('CNP7', 1005, 'Sef aprovizionare');
INSERT INTO persclienti VALUES ('CNP8', 1006, 'Director financiar');
INSERT INTO persclienti VALUES ('CNP9', 1007, 'Sef aprovizionare');

INSERT INTO produse VALUES (1, 'Produs 1','buc', 'Tigari', .19);
INSERT INTO produse VALUES (2, 'Produs 2','kg', 'Bere', .09);
INSERT INTO produse VALUES (3, 'Produs 3','kg', 'Bere', .19);
INSERT INTO produse VALUES (4, 'Produs 4','l', 'Dulciuri', .09);
INSERT INTO produse VALUES (5, 'Produs 5','buc', 'Tigari', .19);
INSERT INTO produse VALUES (6, 'Produs 6','p250g', 'Cafea', .19);

INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1111, DATE'2007-08-01', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs) VALUES (1112, DATE'2007-08-01', 1005,
    'Probleme cu transportul');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1113, DATE'2007-08-01', 1002);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1114, DATE'2007-08-01', 1006);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1115, DATE'2007-08-02', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs) VALUES (1116, DATE'2007-08-02', 1007,
    'Pretul propus initial a fost modificat');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1117, DATE'2007-08-03', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1118, DATE'2007-08-04', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1119, DATE'2007-08-07', 1003);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1120, DATE'2007-08-07', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1121, DATE'2007-08-07', 1004);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1122, DATE'2007-08-07', 1005);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2111, DATE'2007-08-14', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs)
    VALUES (2112, DATE'2007-08-14', 1005, 'Probleme cu transportul');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2113, DATE'2007-08-14', 1002);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2115, DATE'2007-08-15', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs)
    VALUES (2116, DATE'2007-08-15', 1007, 'Pretul propus initial a fost modificat');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2117, DATE'2007-08-16', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2118, DATE'2007-08-16', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2119, DATE'2007-08-21', 1003);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2121, DATE'2007-08-21', 1004);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2122, DATE'2007-08-22', 1005);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3111, DATE'2007-09-01', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs)
    VALUES (3112, DATE'2007-09-01', 1005, 'Probleme cu transportul');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3113, DATE'2007-09-02', 1002);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3115, DATE'2007-09-02', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs)
    VALUES (3116, DATE'2007-09-10', 1007, 'Pretul propus initial a fost modificat');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3117, DATE'2007-09-10', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3118, DATE'2007-09-17', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3119, DATE'2007-10-07', 1003);

INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,1,1,50,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,2,2,75,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,3,5,500,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1112,1,2,80,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1112,2,3,40,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1113,1,2,100,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,1,2,70,1070);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,2,4,30,1705);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,3,5,700,7064);
```

```
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1115,1,2,150,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1116,1,2,125,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1117,1,2,100,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1117,2,1,100,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1118,1,2,30,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1118,2,1,150,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,3,4,50,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,4,5,750,6300);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1120,1,2,80,1120);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1121,1,5,550,7064);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1121,2,2,100,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,1,1,57,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,2,2,79,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,3,5,510,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2112,1,2,85,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2112,2,3,65,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2113,1,2,120,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2115,1,2,110,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2116,1,2,135,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2117,1,2,150,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2117,2,1,110,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2118,1,2,39,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2118,2,1,120,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,3,4,55,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,4,5,755,6300);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2121,1,5,550,7064);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2121,2,2,103,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,1,1,57,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,2,2,79,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,3,5,510,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3112,1,2,85,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3112,2,3,65,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3113,1,2,120,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3115,1,2,110,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3116,1,2,135,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3117,1,2,150,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3117,2,1,110,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3118,1,2,39,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3118,2,1,120,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,3,4,55,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,4,5,755,6300);

INSERT INTO incasari VALUES (1234, DATE'2007-08-15', 'OP', '111', DATE'2007-08-10' );
INSERT INTO incasari VALUES (1235, DATE'2007-08-15','CHIT', '222', DATE'2007-08-15');
INSERT INTO incasari VALUES (1236, DATE'2007-08-16', 'OP', '333', DATE'2007-08-09');
INSERT INTO incasari VALUES (1237, DATE'2007-08-17', 'CEC', '444', DATE'2007-08-10');
INSERT INTO incasari VALUES (1238, DATE'2007-08-17', 'OP', '555', DATE'2007-08-10');
INSERT INTO incasari VALUES (1239, DATE'2007-08-18', 'OP', '666', DATE'2007-08-11');

INSERT INTO incasfact VALUES (1234, 1111, 53996);
INSERT INTO incasfact VALUES (1234, 1118, 101975);
INSERT INTO incasfact VALUES (1235, 1112, 125516);
INSERT INTO incasfact VALUES (1236, 1117, 9754);
```

```
INSERT INTO incasfact VALUES (1236, 1118, 100000);
INSERT INTO incasfact VALUES (1236, 1120, 7315);
INSERT INTO incasfact VALUES (1237, 1117, 9754);
INSERT INTO incasfact VALUES (1238, 1113, 106275);
INSERT INTO incasfact VALUES (1239, 1117, 3696);

COMMIT;
```

3.4.6. Scripturile de populare în PostgreSQL

PostgreSQL-ul este de-a dreptul interesant. Vă amintiţi (listingul 3.5) că, pentru atributele IdDoctor din tabela DOCTORI şi IdPacient din tabela PACIENŢI, am recurs la tipul SERIAL, iar pentru IdExaminare din tabela TRIAJ la tipul BIGSERIAL[14]. Ne-am aştepta, astfel, ca lansarea scriptului Oracle (listing 3.11) în PostgreSQL să se soldeze cu problemele întâlnite în DB2 pentru coloanele autogenerate, întrucât comenzile INSERT conţin valori pentru atributele SERIAL/BIGSERIAL. Ei bine, PostgreSQL-ul nu are nicio obiecţie! Mai mult, nici dacă lansăm scriptul din listingul 3.11 de mai multe ori nu recepţionăm nicio eroare. Explicaţia constă în faptul că pentru PostgreSQL, valorile „seriale" depind decisiv de liniile existente în tabelă, iar o valoare specificată în comanda INSERT suprascrie valoarea implicită. Deşi pare o binecuvântare, această relaxare poate genera însă probleme destul de neplăcute, cum ar fi violarea ulterioară a cheii primare, prin suprapunerea următoarei valori autogenerate cu o valoare anterior introdusă printr-o comandă INSERT. Este motivul pentru care în tabelele BD VÂNZĂRI nu am mai recurs la tipurile SERIAL şi BIGSERIAL.

Listingul de populare a acestor tabele este identic cu cel din Oracle (3.12), chiar dacă, similar DB2, PostgreSQL permite inserarea mai multor înregistrări cu o singură comandă INSERT:

```
INSERT INTO judete VALUES
    ('IS', 'Iasi', 'Moldova'), ('VN', 'Vrancea', 'Moldova'), ('NT', 'Neamt', 'Moldova'),
    ('SV', 'Suceava', 'Moldova'), ('VS', 'Vaslui', 'Moldova'), ('TM', 'Timis', 'Banat');
```

3.4.7. Scripturile de populare în SQL Server

SQL Server este, ca şi DB2, scrupulos în privinţa atributelor IDENTITY. Dacă în DB2 funcţionează varianta *INSERT INTO doctori VALUES (DEFAULT, 'Vasilcu Ionel', 'chirurgie', '1965-11-11')*, în SQL Server nu, sintaxa acceptată fiind:

```
INSERT INTO doctori (numedoctor, specialitate, datanasterii)
    VALUES ('Vasilcu Ionel', 'chirurgie', '1965-11-11');
```

De asemenea, execuţii repetate ale scriptului de populare care începe cu golirea tabelelor (DELETE FROM...) consumă valorile autoincrementate, astfel încât apar probleme cu specificarea valorilor pentru cheile străine. Scriptul de populare SQL Server a tabelelor din BD TRIAJ este prezentat în listing 3.13.

14. Nici SERIAL, nici BIGSERIAL nu sunt tipuri propriu-zise.

Listing 3.13. Script MS SQL Server de populare a tabelelor BD TRIAJ

```
DELETE FROM triaj ; DELETE FROM pacienti;
DELETE FROM garzi ; DELETE FROM doctori;

INSERT INTO doctori (numedoctor, specialitate,datanasterii)
    VALUES ('Vasilcu Ionel', 'chirurgie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate,datanasterii)
    VALUES ('Georgescu Mircea', 'hepatologie', '1966-01-12');
INSERT INTO doctori (numedoctor, specialitate,datanasterii)
    VALUES ('Zahir Tudorel', 'chirurgie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate,datanasterii)
    VALUES ('Bostan Vasile', 'cardiologie', '1965-11-11');
INSERT INTO doctori (numedoctor, specialitate,datanasterii)
    VALUES ('Popescu Ionela', 'boli interne', '1965-11-11');

INSERT INTO garzi VALUES (1, '2008-01-03 7:00:00', '2008-01-03 19:00:00');
INSERT INTO garzi VALUES (2, '2008-01-03 19:00:00', '2008-01-04 7:00:00');
INSERT INTO garzi VALUES (3, '2008-01-04 07:00:00', '2008-01-04 18:00:00');
INSERT INTO garzi VALUES (4, '2008-01-04 18:00:00', '2008-01-05 06:00:00');
INSERT INTO garzi VALUES (1, '2008-01-05 6:00:00', '2008-01-05 14:00:00');
INSERT INTO garzi VALUES (2, '2008-01-05 14:00:00', '2008-01-05 22:00:00');
INSERT INTO garzi VALUES (5, '2008-01-05 22:00:00', '2008-01-06 9:00:00');

INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Stroe Mihaela', 2601228390834, 'Bd. Cantemir, 32, Bl.G4, Sc.C, Ap.4', 'MR366766');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Buzatu Corneliu', 1650512370514, 'Str. Desprimaveririi, 112', 'MX456783');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Spineanu Marius', 5010101380625, 'Bd. Stefan cel Mare, 4, Bl.I1, Sc.A, Ap.24',
      'MX213345');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Bagdasar Adela', 2601002250611, 'Str. Primaverii, 17', 'MX345678');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Ionascu Ionela', 6001122390199, 'Str. Florilor', 'MX654322');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Popescu Maria-Mirabela', 2721231300888, 'Bd. 22 Decembrie, 2, Bl.5, Sc.B, Ap.21',
      'MX765432');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Stroescu Mihaela Oana', 6020719120545, 'Str. Lenei Nr. 234', 'MX876567');
INSERT INTO pacienti (numepacient, CNP, adresa, serie_nr_act_identitate)
    VALUES ('Cazan Ana Maria', 2690202200345, 'Bd. Independentei, 87, Bl.K3, Sc.A, Ap.34',
      'MX987789');

INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 7:18:00', 1, 'dureri de stomac intense', NULL, 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 8:45:00', 2, 'febra puternica, varsaturi', 'penicilina', 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 12:45:00', 3, 'deranjament stomacal', 'scobutil', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-03 20:45:00', 4, 'palpitatii cardiace', 'linistin', 'cardiologie');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 1:28:00', 5, 'plaga profunda picior drept', 'antibiotice, pansament', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 10:45:00', 3, 'contractii stomacale, varsaturi', NULL, 'boli interne');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
    VALUES ('2008-01-04 11:20:00', 7, 'fata de culoare galbena, ameteli', NULL, 'hepatologie');
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
```

```
VALUES ('2008-01-04 22:45:00', 8, 'dureri articulare', 'scobutil', NULL);
INSERT INTO triaj (dataora_examinare, idpacient, simptome, tratament_imediat, sectie_destinatie)
  VALUES ('2008-01-05 6:18:00', 5, 'febra puternica, delir', 'penicilina', 'chirurgie');
```

În privința comenzilor de populare a tabelelor din BD VÂNZĂRI, nu ar fi prea multe de adăugat - vezi listing 3.14 -, doar că pot fi grupate mai multe comenzi pe aceeaşi linie a scriptului, însă comanda INSERT nu permite adăugarea simultană a două sau mai multe linii într-o tabelă.

Listing 3.14. Script MS SQL Server de populare a tabelelor BD VÂNZĂRI

```
DELETE FROM incasfact; DELETE FROM incasari; DELETE FROM liniifact;
DELETE FROM facturi; DELETE FROM produse; DELETE FROM persclienti;
DELETE FROM persoane; DELETE FROM clienti; DELETE FROM coduri_postale;
DELETE FROM judete;

INSERT INTO judete VALUES ('IS', 'Iasi', 'Moldova');
INSERT INTO judete VALUES ('VN', 'Vrancea', 'Moldova');
INSERT INTO judete VALUES ('NT', 'Neamt', 'Moldova');
INSERT INTO judete VALUES ('SV', 'Suceava', 'Moldova');
INSERT INTO judete VALUES ('VS', 'Vaslui', 'Moldova');
INSERT INTO judete VALUES ('TM', 'Timis', 'Banat');

INSERT INTO coduri_postale VALUES ('700505', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('700510', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('700515', 'Iasi', 'IS');
INSERT INTO coduri_postale VALUES ('701150', 'Pascani', 'IS');
INSERT INTO coduri_postale VALUES ('706500', 'Vaslui', 'VS');
INSERT INTO coduri_postale VALUES ('706510', 'Vaslui', 'VS');
INSERT INTO coduri_postale VALUES ('705300', 'Focsani', 'VN');
INSERT INTO coduri_postale VALUES ('705310', 'Focsani', 'VN');
INSERT INTO coduri_postale VALUES ('706400', 'Barlad', 'VS');
INSERT INTO coduri_postale VALUES ('705800', 'Suceava', 'SV');
INSERT INTO coduri_postale VALUES ('705550', 'Roman', 'NT');
INSERT INTO coduri_postale VALUES ('701900', 'Timisoara', 'TM');

INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost, Telefon)
    VALUES ('Client 1 SRL', 'R1001', 'Tranzitiei, 13 bis', '700505', NULL);
INSERT INTO clienti (DenCl, CodFiscal, CodPost, Telefon)
    VALUES ('Client 2 SA', 'R1002', '700505', '0232212121');
INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost, Telefon)
    VALUES ('Client 3 SRL', 'R1003', 'Prosperitatii, 22', '706500','0235222222');
INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost)
    VALUES ('Client 4', 'R1004', 'Sapientei, 56', '701150');
INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost, Telefon)
    VALUES ('Client 5 SRL', 'R1005', NULL, '701900', '0256111111');
INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost, Telefon)
    VALUES ('Client 6 SA', 'R1006', 'Pacientei, 33', '705550', NULL);
INSERT INTO clienti (DenCl, CodFiscal, Adresa, CodPost, Telefon) VALUES
('Client 7 SRL', 'R1007', 'Victoria Capitalismului, 2', '701900', '0256121212');

INSERT INTO persoane VALUES ('CNP1', 'Ioan', 'Vasile', 'I.L.Caragiale, 22', 'M', '700505',
'123456',
     '987654', '094222222', NULL);
INSERT INTO persoane VALUES ('CNP2', 'Vasile', 'Ion', NULL, 'M', '700505', '234567', '876543',
     '094222223', 'Ion@a.ro');
INSERT INTO persoane VALUES ('CNP3', 'Popovici', 'Ioana', 'V.Micle, Bl.I, Sc.B,Ap.2', 'F',
'701150',
```

```
        '345678', NULL, '094222224', NULL);
    INSERT INTO persoane VALUES ('CNP4', 'Lazar', 'Caraion', 'M.Eminescu, 42', 'M', '706500',
        '456789', NULL, '094222225', NULL);
    INSERT INTO persoane VALUES ('CNP5', 'Iurea', 'Simion', 'I.Creanga, 44 bis', 'M', '706500',
        '567890', '543210', NULL, NULL);
    INSERT INTO persoane VALUES ('CNP6', 'Vasc', 'Simona', 'M.Eminescu, 13', 'F', '701150', NULL,
        '432109', '094222227', NULL);
    INSERT INTO persoane VALUES ('CNP7', 'Popa', 'Ioanid', 'I.Ion, Bl.H2, Sc.C, Ap.45', 'M', '701900',
        '789012', '321098', NULL, NULL);
    INSERT INTO persoane VALUES ('CNP8', 'Bogacs', 'Ildiko', 'I.V.Viteazu, 67', 'F', '705550',
'890123',
        '210987', '094222229', NULL);
    INSERT INTO persoane VALUES ('CNP9', 'Ioan', 'Vasilica', 'Garii, Bl.B4, Sc.A, Ap.1', 'F', '701900',
        '901234', '109876', '094222230', NULL);

    INSERT INTO persclienti VALUES ('CNP1', 1001, 'Director general');
    INSERT INTO persclienti VALUES ('CNP2', 1002, 'Director general');
    INSERT INTO persclienti VALUES ('CNP3', 1002, 'Sef aprovizionare');
    INSERT INTO persclienti VALUES ('CNP4', 1003, 'Sef aprovizionare');
    INSERT INTO persclienti VALUES ('CNP5', 1003, 'Director financiar');
    INSERT INTO persclienti VALUES ('CNP6', 1004, 'Director general');
    INSERT INTO persclienti VALUES ('CNP7', 1005, 'Sef aprovizionare');
    INSERT INTO persclienti VALUES ('CNP8', 1006, 'Director financiar');
    INSERT INTO persclienti VALUES ('CNP9', 1007, 'Sef aprovizionare');

    INSERT INTO produse VALUES ('Produs 1','buc', 'Tigari', .19);
    INSERT INTO produse VALUES ('Produs 2','kg', 'Bere', .09);
    INSERT INTO produse VALUES ('Produs 3','kg', 'Bere', .19);
    INSERT INTO produse VALUES ('Produs 4','l', 'Dulciuri', .09);
    INSERT INTO produse VALUES ('Produs 5','buc', 'Tigari', .19);
    INSERT INTO produse VALUES ('Produs 6','p250g', 'Cafea', .19);

    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1111, '2007-08-01', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (1112, '2007-08-01', 1005,
        'Probleme cu transportul');
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1113, '2007-08-01', 1002);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1114, '2007-08-01', 1006);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1115, '2007-08-02', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (1116, '2007-08-02', 1007,
        'Pretul propus initial a fost modificat');
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1117, '2007-08-03', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1118, '2007-08-04', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1119, '2007-08-07', 1003);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1120, '2007-08-07', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1121, '2007-08-07', 1004);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (1122, '2007-08-07', 1005);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2111, '2007-08-14', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (2112, '2007-08-14', 1005,
        'Probleme cu transportul');
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2113, '2007-08-14', 1002);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2115, '2007-08-15', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (2116, '2007-08-15', 1007,
        'Pretul propus initial a fost modificat');
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2117, '2007-08-16', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2118, '2007-08-16', 1001);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2119, '2007-08-21', 1003);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2121, '2007-08-21', 1004);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (2122, '2007-08-22', 1005);
    INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3111, '2007-09-01', 1001);
```

```
INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (3112, '2007-09-01', 1005,
    'Probleme cu transportul');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3113, '2007-09-02', 1002);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3115, '2007-09-02', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) VALUES (3116, '2007-09-10', 1007,
    'Pretul propus initial a fost modificat');
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3117, '2007-09-10', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3118, '2007-09-17', 1001);
INSERT INTO facturi (NrFact, DataFact, CodCl) VALUES (3119, '2007-10-07', 1003);

INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,1,1,50,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,2,2,75,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1111,3,5,500,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1112,1,2,80,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1112,2,3,40,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1113,1,2,100,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,1,2,70,1070);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,2,4,30,1705);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1114,3,5,700,7064);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1115,1,2,150,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1116,1,2,125,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1117,1,2,100,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1117,2,1,100,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1118,1,2,30,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1118,2,1,150,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,3,4,50,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1119,4,5,750,6300);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1120,1,2,80,1120);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1121,1,5,550,7064);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (1121,2,2,100,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,1,1,57,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,2,2,79,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2111,3,5,510,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2112,1,2,85,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2112,2,3,65,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2113,1,2,120,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2115,1,2,110,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2116,1,2,135,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2117,1,2,150,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2117,2,1,110,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2118,1,2,39,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2118,2,1,120,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,3,4,55,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2119,4,5,755,6300);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2121,1,5,550,7064);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (2121,2,2,103,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,1,1,57,1000);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,2,2,79,1050);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3111,3,5,510,7060);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3112,1,2,85,1030);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3112,2,3,65,750);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3113,1,2,120,975);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3115,1,2,110,925);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3116,1,2,135,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3117,1,2,150,1000);
```

```
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3117,2,1,110,950);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3118,1,2,39,1100);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3118,2,1,120,930);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,1,2,35,1090);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,2,3,40,700);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,3,4,55,1410);
INSERT INTO liniifact (NrFact,Linie,CodPr,Cantitate,PretUnit) VALUES (3119,4,5,755,6300);

INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-15', 'OP', '111', '2007-08-10' );
INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-15', 'CHIT', '222', '2007-08-15');
INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-16', 'OP', '333', '2007-08-09');
INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-17', 'CEC', '444', '2007-08-10');
INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-17', 'OP', '555', '2007-08-10');
INSERT INTO incasari (DataInc,CodDoc,NrDoc, DataDoc) VALUES
    ('2007-08-18', 'OP', '666', '2007-08-11');

INSERT INTO incasfact VALUES (1234, 1111, 53996);
INSERT INTO incasfact VALUES (1234, 1118, 101975);
INSERT INTO incasfact VALUES (1235, 1112, 125516);
INSERT INTO incasfact VALUES (1236, 1117, 9754);
INSERT INTO incasfact VALUES (1236, 1118, 100000);
INSERT INTO incasfact VALUES (1236, 1120, 7315);
INSERT INTO incasfact VALUES (1237, 1117, 9754);
INSERT INTO incasfact VALUES (1238, 1113, 106275);
INSERT INTO incasfact VALUES (1239, 1117, 3696);
```

3.5. Tranzacţii

În câteva dintre scripturile de populare a tabelelor, prezentate în paragrafele anterioare, a fost strecurată, pe final, o comandă discretă - COMMIT. Şi în limba română, mai ales în jargonul orăşenesc, a apărut termenul „a comis-o !", în sensul că a gafat-o fără dubii/întoarcere. Oricât ar părea de ciudat, termenul se învecinează cu mai sus amintita comandă.

3.5.1. Principiile de bază ale tranzacţiilor

După cum afirmam în capitolul 1, tranzacţiile grupează comenzi care operează pe principiul *toţi pentru unul, unul pentru toţi*. Comenzile incluse în tranzacţii sunt de tip DML (INSERT, UPDATE sau DELETE). Începutul tranzacţiei poate fi implicit (în Oracle, prima comandă DML dintr-o sesiune de lucru marchează, automat, începutul unei tranzacţii; la fel, prima comandă care urmează sfârşitului unei tranzacţii) sau explicit, prin comanda BEGIN TRANSACTION sau START TRANSACTION. Încheierea cu succes a unei tranzacţii este marcată prin comanda

COMMIT[15]. Încheierea eronată şi, implicit, anularea efectului tuturor comenzilor de la începutul tranzacţiei până în momentul curent se realizează prin ROLLBACK.

Să luăm cazul cel mai des invocat atunci când se încearcă explicarea principiilor de bază ale unei tranzacţii: plata de la un bancomat a facturii telefonice. Plătitorul are un cont - ContPlătitor la banca X. Plata trebuie făcută în contul operatorului de telefonie - ContBeneficiar deschis la banca Y (X poate fi egal cu Y, dar asta nu are importanţă pentru discuţia noastră). Iată scenariul:

- autentificarea plătitorului (introducere card, apoi tastare PIN);
- alegerea, folosind meniul bancomatului, a operaţiunii, apoi tastarea sumei şi a contului client; pe baza acestor elemente, aplicaţia identifică ContPlătitor şi ContBeneficiar;
- se adaugă în baza de date în care se află ContPlătitor înregistrări care consemnează momentul plăţii, bancomatul de la care s-a făcut plata etc.;
- se scade Suma din ContPlătitor;
- se adaugă în baza de date în care se află ContBeneficiar informaţii privind plata (data/ora încasării, modul de încasare, sursa banilor - ContPlătitor) etc.;
- se adună Suma la ContBeneficiar;
- se eliberează chitanţa.

Tranzacţia propriu-zisă începe cu pasul 3 şi se finalizează cu pasul 6. Trecerea la pasul următor este condiţionată de confirmarea pasului curent. Dacă oricare din paşii 3-6 generează o eroare (se pierde legătura cu serverul pe care este stocat ContPlătitor sau suma disponibilă în ContPlătitor este mai mică decât suma de plătit etc.), atunci întreaga tranzacţie este abandonată. Dacă, însă, şi pasul 6 a decurs cu succes, tranzacţia este încheiată cu succes.

Cea mai tristă ar fi situaţia în care Suma este scăzută din ContPlătitor, dar nu poate fi adăugată în ContBeneficiar (a picat reţeaua de comunicaţii!). Plătitorul ar avea conştiinţa curată (din punctul de vedere al plăţii), în timp ce pentru operatorul de telefonie ar fi un rău platnic. Ei bine, gruparea paşilor în tranzacţii ar trebui să prevină asemenea gen de probleme.

Pentru a vă face o idee asupra importanţei tranzacţiilor, trebuie să avem în vedere că toţi aceşti paşi trebuie să se deruleze în condiţiile în care, simultan, accesează şi modifică baza de date un număr foarte mare de utilizatori. Să presupunem că plătitorul are un acord cu banca, prin care plata facturii la gaz metan se face automat la o anumită dată într-un anumit cont - ContBeneficiar2. Este posibil ca, tocmai în momentele în care posesorul cardului meştereşte la bancomat, încercând să-şi plătească factura la telefon, aplicaţia de plată automată a facturii la gaz să debiteze ContPlătitor, după un scenariu similar celui de mai sus. Ei bine, dacă prima tranzacţie (cea cu plata facturii telefonice) este începută, a doua

15. Unele servere BD (cum este Oracle) încheie (cu succes) o tranzacţie, implicit, la execuţia unei comenzi DDL.

tranzacţie nu „vede" nicio modificare în soldul ContPlătitor până în momentul finalizării cu succes a primeia[16].

În standardele SQL, comenzile legate de controlul tranzacţiilor sunt:

- START TRANSACTION permite pornirea şi, eventual, configurarea unei tranzacţii;
- SET TRANSACTION stabileşte parametrii următoarei tranzacţii pentru sesiunea curentă;
- SET CONSTRAINTS stabileşte regimul restricţiilor pentru tranzacţia curentă (dacă este începută) sau pentru următoarea/următoarele (când nu există o tranzacţie curentă);
- SAVEPOINT declară un punct de salvare; într-o tranzacţie pot fi mai multe puncte de salvare, pentru ca, în caz de erori, să nu piardă efectul tuturor comenzilor din tranzacţie, ci până la un anumit SAVEPOINT;
- RELEASE SAVEPOINT anulează un punct de salvare;
- COMMIT încheie (cu succes) tranzacţia curentă;
- ROLLBACK încheie (cu eşec) tranzacţia curentă, refăcând starea BD dinaintea începerii tranzacţiei curente.

Dintre cele patru servere, DB2 şi Oracle prezintă tranzacţii implicite, în timp în PostgreSQL tranzacţiile trebuie declarate prin comanda BEGIN sau prin comanda START TRANSACTION. SQL Server permite şi alocarea unui nume pentru fircare tranzacţie. Tranzacţiile explicite încep cu BEGIN TRANSACTION şi se încheie fie prin COMMIT TRANSACTION sau ROLLBACK TRANSACTION.

3.5.2. Restricţii amânabile

Restricţiile declarate într-o tabelă sunt verificate imediat după o comandă INSERT sau UPDATE. Lucrurile pot fi, însă, uşor modificate. Să revenim la cazul facturilor, cu mici schimbări. Listingul 3.15 creează două tabele, EX_FACT şi EX_FACT_DETALII, care sunt versiuni *light* ale tabelelor FACTURI şi LINIIFACT. Sintaxa este cea a dialectului SQL din Oracle.

Listing 3.15. Script Oracle de testare a unor restricţii neamânabile

```
DROP TABLE ex_fact_detalii;
DROP TABLE ex_fact;

CREATE TABLE ex_fact (
    NrFact NUMBER(6) NOT NULL PRIMARY KEY,
    DataFact DATE DEFAULT CURRENT_DATE NOT NULL,
    NrLinii NUMBER(2) DEFAULT 0 NOT NULL
        CONSTRAINT ck_linii CHECK (NrLinii > 0 )
    );

CREATE TABLE ex_fact_detalii (
    NrFact NUMBER(6) NOT NULL REFERENCES ex_fact (NrFact),
```

16. În manualele de baze de date se pune mare preţ pe problema gestiunii tranzacţiilor şi a simultaneităţii (concurenţei). Pentru o tratare *in extenso*, inclusiv cele patru proprietăţi (ACID) şi nivelele de izolare, vezi [Dollinger 1998], pp. 183-254, [Ionescu 2004], pp. 237-280.

```
Linie NUMBER(2) NOT NULL CONSTRAINT ck_linie_fd CHECK (Linie > 0),
Produs VARCHAR2(40) NOT NULL,
Cantitate NUMBER(8) NOT NULL,
PretUnit NUMBER (9,2) NOT NULL,
CONSTRAINT pk_fact_detalii PRIMARY KEY (NrFact,Linie),
CONSTRAINT un_fact_detalii UNIQUE (NrFact, Produs)
);
```

În tabela EX_FACT există, printre altele, un atribut calculat care „spune" câte linii are factura respectivă[17]. Interesul nostru este să nu existe facturi fără nicio linie (în FACTURI există asemenea situaţii nedorite), astfel încât am definit restricţia CK_LINII potrivit căreia valoarea atributului NrLinii trebuie să fie strict mai mare ca zero. Cum e şi normal, atunci când inserăm prima înregistrare în tabela EX_FACT: *INSERT INTO ex_fact (NrFact) VALUES (123456)*, ne pricopsim cu un mesaj de eroare - vezi figura 3.9 - întrucât valoarea implicită a NrLinii este 0.

Modificăm tipul restricţiei CK_LINII, adăugând în comanda de creare - vezi listing 3.16 - două clauze, DEFERRABLE şi INITIALLY DEFERRED. Prima semnalizează că restricţia cu pricina este *amânabilă*, adică poate fi amânată până la finalul tranzacţiei. Cealaltă clauză indică faptul că restricţia este nu numai amânabilă, dar şi *amânată*. Aceasta deoarece, ulterior, dacă vrem să trecem o restricţie pe starea „clasică" (verificare imediată, adică chiar în momentul INSERT-ului sau UPDATE-ului), folosim clauza *SET CONSTRAINT ck_linii IMMEDIATE*.

Vă întrebaţi, probabil, de ce a fost nevoie să ştergem şi apoi să recreăm tabelele. Ei bine, am fost forţaţi. În Oracle, o restricţie neamânabilă nu poate fi modificată în amânabilă.

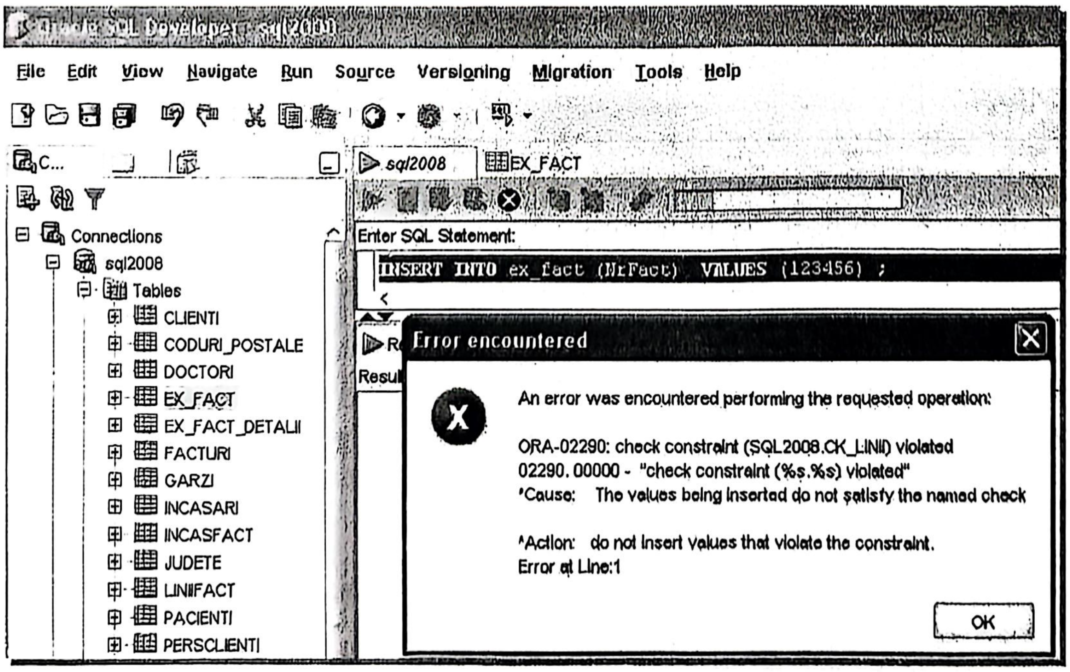


Figura 3.9. Violarea restricţiei (neamânabile) CK_LINII

17. Controlul valorii acestui atribut este posibil prin declanşatoare - vezi capitolul 17.

Listing 3.16. Script Oracle de testare a unei restricţii amânabile/amânate

```
DROP TABLE ex_fact_detalii;
DROP TABLE ex_fact;

CREATE TABLE ex_fact (
    NrFact NUMBER(6) NOT NULL PRIMARY KEY,
    DataFact DATE DEFAULT CURRENT_DATE NOT NULL,
    NrLinii NUMBER(2) DEFAULT 0 NOT NULL
        CONSTRAINT ck_linii CHECK (NrLinii > 0 ) DEFERRABLE INITIALLY DEFERRED
    );

CREATE TABLE ex_fact_detalii (
    NrFact NUMBER(6) NOT NULL REFERENCES ex_fact (NrFact),
    Linie NUMBER(2) NOT NULL CONSTRAINT ck_linie_fd CHECK (Linie > 0),
    Produs VARCHAR2(40) NOT NULL,
    Cantitate NUMBER(8) NOT NULL,
    PretUnit NUMBER (9,2) NOT NULL,
    CONSTRAINT pk_fact_detalii PRIMARY KEY (NrFact,Linie),
    CONSTRAINT un_fact_detalii UNIQUE (NrFact, Produs)
    );

INSERT INTO ex_fact (NrFact) VALUES (123456);

INSERT INTO ex_fact_detalii VALUES (123456, 1, 'Un produs', 100, 50);

UPDATE ex_fact SET NrLinii = NrLinii + 1 WHERE NrFact = 123456;

COMMIT;
```

Comenzile INSERT şi UPDATE din script funcţionează fără probleme. Dacă, însă, uităm comanda UPDATE care incrementează NrLinii, la execuţia comenzii COMMIT se generează eroarea din figura 3.10.

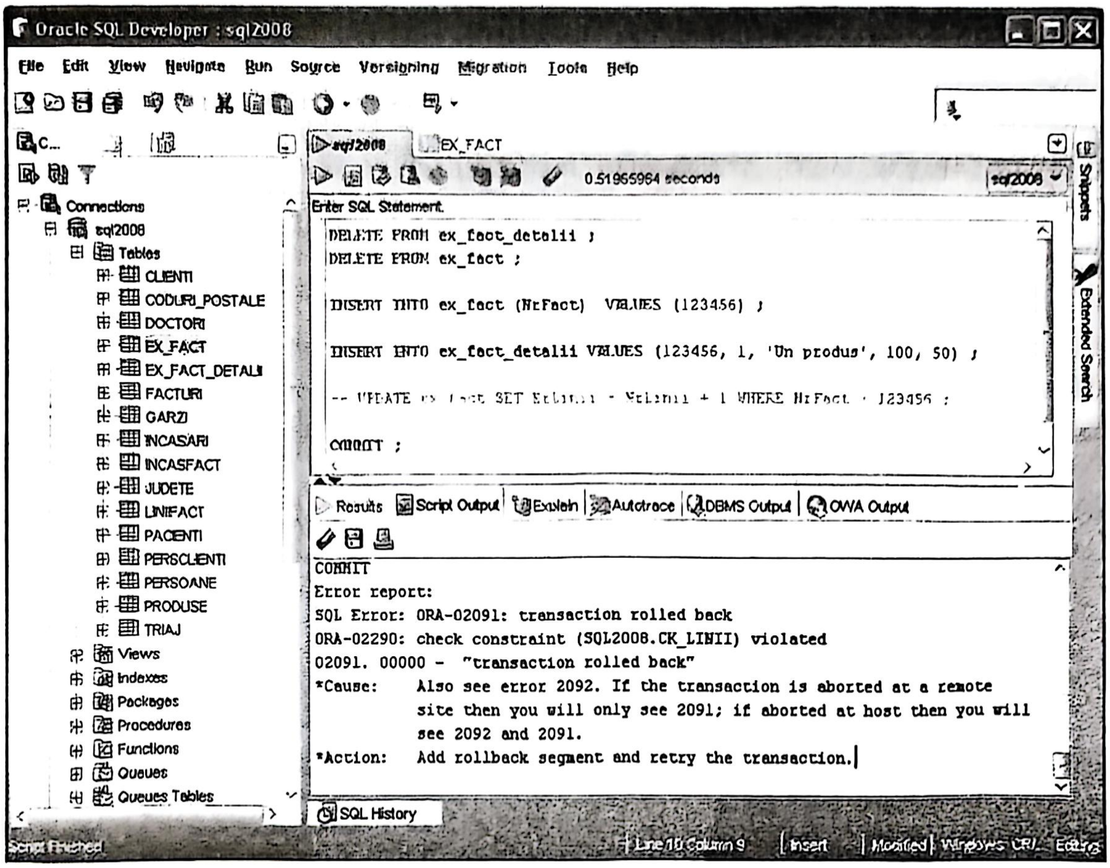


Figura 3.10. Violarea restricției (amânabile) CK_LINII

Nu întâmplător am exemplificat restricțiile amânabile în Oracle. Nici DB2 şi nici SQL Server nu au implementat mecanismul, iar în PostgreSQL doar restricțiile referențiale pot fi amânate.

3.5.3. Tabele temporare globale şi locale

O tabelă temporară, globală sau locală are o durată de viață limitată fie la o sesiune de lucru, fie la o tranzacție. Detalii despre comenzile de creare şi opțiunile de lucru cu tabelele temporare sunt prezentate în capitolul 14.

Capitolul 4
Limbaje de interogare. Algebra relațională

Dacă un model de date are, potrivit lui Codd, trei axe - *structură* (modalitatea de organizare a datelor în bază), *integritate* (restricții) și *manipulare* (operații), până în acest capitol discuția s-a purtat la nivelul primelor două. Este momentul să trecem la partea cea mai importantă a cărții de față - modalitățile prin care se pot extrage informații dintr-o bază de date.

În materie de limbaje dedicate bazelor de date, SQL-ul s-a instalat, de aproape trei decenii, confortabil în postura de lider absolut. Lucrurile nu au fost atât de clare de la început, peisajul limbajelor de interogare fiind mult mai bogat decât ne-am fi imaginat. Mai mult, se spune uneori că elementul-cheie în impunerea SQL-ului în detrimentul altor limbaje, cum ar Quel, QBE, nu a fost forța acestuia, ci faptul că a fost susținut de cine trebuia (IBM, la care s-a adăugat ulterior Oracle), plus faptul că s-a reuşit elaborarea unor standarde unanim acceptate.

Inițial, pentru exprimarea operațiilor aplicabile structurilor de date, E.F.Codd, părintele relaționalului, a definit un limbaj de manipulare a datelor puternic matematizat, bazat pe teoria ansamblurilor (seturilor), numit DSL/Alpha (vezi și [Date1999-1]). Tot Codd pune bazele algebrei relaționale - limbaj tratat în continuare - care, deşi teoretică, constituie fundamentul SQL-ului, chiar dacă astăzi puțini mai sunt interesați de această paternitate. Am putea spune că SQL-ul este, în oarecare măsură, fiul deopotrivă rătăcitor şi ne(re)cunoscător al algebrei relaționale.

4.1. Caracterizare generală a limbajelor de interogare

Până la consacrarea SGBD-urilor relaționale, extragerea informațiilor dorite dintr-o bază de date se realiza prin aplicații dezvoltate exclusiv cu limbaje procedurale, în care trebuie definite atât datele dorite, cât şi metodele de căutare şi extragere ale acestora. Succesul SGBDR-urilor ar fi fost de neconceput fără elaborarea şi implementarea unor limbaje performante pentru manipularea BD - limbajele de interogare.

Limbajele relaționale sunt neprocedurale: utilizatorul defineşte datele ce trebuie extrase din BD, sarcina căutării şi extragerii fiind „rezervată" exclusiv SGBD-ului. De asemenea, în unele lucrări, acestea sunt referite ca limbaje închise, deoarece o consultare generează o nouă relație ce poate fi utilizată, la rândul său, ca argument în alte consultări ş.a.m.d.

Pornind de la cele două modalităţi de definire a relaţiei, pe de o parte, ca predicat aplicat asupra unor domenii şi, pe de altă parte, ca ansamblu de tupluri, limbajele de manipulare a datelor sunt grupate în două mari categorii, limbaje *predicative* - fondate pe teoria predicatelor şi limbaje *ansambliste* - fondate pe teoria ansamblurilor (tuplurilor). Limbajele predicative sunt divizate în alte două sub-clase, cele care au la bază *calculul relaţional asupra tuplurilor* (limbaje orientate pe tupluri) şi cele în care *calculul relaţional se aplică asupra domeniilor* (limbaje orientate pe domenii). Elementul definitoriu pentru limbajele de manipulare bazate pe calculul predicatelor îl reprezintă noţiunea de variabilă, care poate fi asociată fie tuplurilor, fie domeniilor.

O altă grupare delimitează limbajele *non-grafice* de cele *grafice*. Primele permit reprezentarea unei consultări „în linie", prin dispunerea succesivă a operatorilor, atributelor şi relaţiilor. Cele grafice permit redactarea consultării în mod interactiv, prin afişarea pe ecran a unui sistem de meniuri şi elemente de dialog din care opţiunile pot fi selectate şi modificate cu ajutorul mouse-ului (sau claviaturii); în plus, se mai poate opera o delimitare şi pentru limbajele grafice în funcţie de utilizarea explicită sau implicită a variabilelor de domeniu. Deşi nu exagerat de recentă, o clasificare încă valabilă (în parte) a limbajelor relaţionale poate fi prezentată ca în figura 4.1.

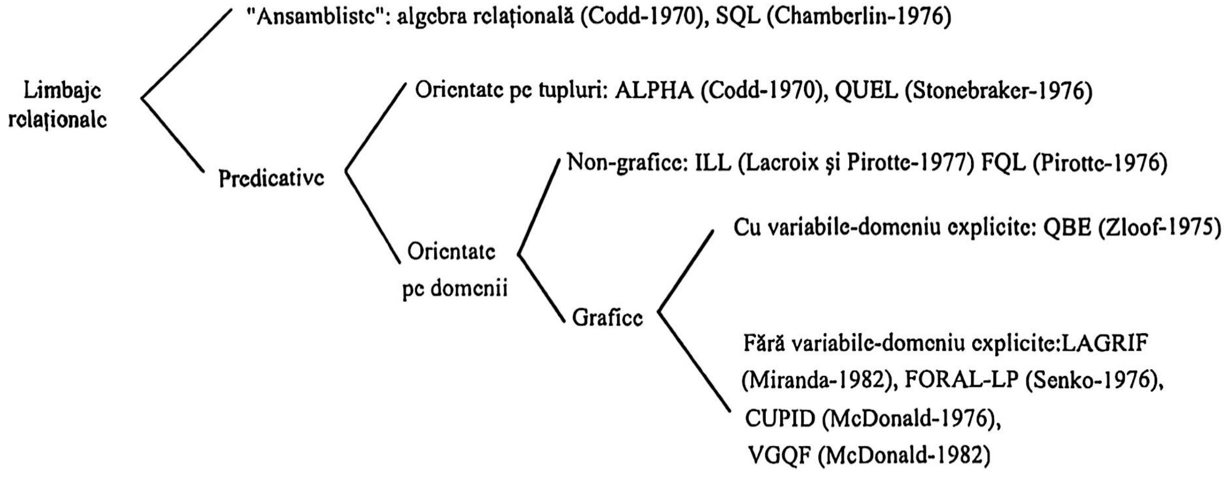


Figura 4.1. O clasificare (mai veche) a limbajelor relaţionale

Există o serie de caracteristici comune tuturor limbajelor:

- operatorii relaţionali se aplică relaţiilor luate în întregime, adică tuturor tuplurilor care alcătuiesc relaţiile respective;
- rezultatul fiecărui operator (rezultatul consultării) este o nouă relaţie ce poate servi ca argument într-o altă consultare ş.a.m.d.;
- logica operatorilor se bazează pe valorile atributelor, ceea ce constituie de altminteri suportul singurului mod de acces în BD; întrucât consultările mono şi multirelaţii sunt efectuate exclusiv prin compararea valorilor atributelor (definite pe domenii compatibile), accesul total independent de limbaj este asigurat.

Înaintea redactării unei consultări într-un limbaj relațional trebuie parcursă o fază de *analiză*, pentru determinarea atributelor rezultatului, legăturilor dintre tabele şi eventualelor condiții (restricții) ce trebuie respectate.

Algebra şi calculul relațional servesc drept puncte de referință în caracterizarea unui limbaj ca fiind *complet* sau incomplet, din punct de vedere relațional. Dacă un limbaj permite exprimarea tuturor operatorilor enumerați mai jos, oferind măcar facilitățile algebrei relaționale, se spune că acesta este un limbaj relațional complet ([Saleh 1994], p. 35).

4.2. Operatorii algebrei relaționale

Limbajul algebric relațional cuprinde două tipuri de operatori: *ansamblişti* - REUNIUNE, INTERSECȚIE, DIFERENȚĂ, PRODUS CARTEZIAN - şi *relaționali* - SELECȚIE, PROIECȚIE, JONCȚIUNE şi DIVIZIUNE.

O altă clasificare distinge operatorii *fundamentali*, ireductibili (reuniunea, diferența, produsul cartezian, selecția, proiecția) de cei *derivați*, a căror funcționalitate poate fi realizată prin combinarea operatorilor fundamentali (intersecția, joncțiunea, diviziunea).

Pentru cele ce urmează se notează cu:

- *t* sau *r*, un *tuplu* al unei relații (linie a unei tabele);
- *t(A)*, un *subtuplu* al relației R, relativ la atributul A (valoarea atributului A în linia *t*).

Trei dintre operatorii ansamblişti - *reuniunea* („$\cup$"), *intersecția* („$\cap$") şi *diferența* („-") - pot opera numai cu două relații *unicompatibile*. Fie R1 (A_1, A_2, ..., A_n) şi R2 (B_1, B_2, ..., B_m) două relații. R_1 şi R_2 sunt unicompatibile dacă: a) n = m; b) $\forall$ i $\in$ {1,2, ..., n}, A_i şi B_i sunt de acelaşi tip sintactic (aceasta nu înseamnă că trebuie să prezinte, neapărat, domenii identice de definire). Relațiile R1 şi R2 din figura 5.1 sunt unicompatibile deoarece: a) ambele au acelaşi număr de atribute; b) atributele A, B, C din R1 (le putem nota şi R1.A, R1.B, R1.C) corespund sintactic (sunt de acelaşi tip) atributelor C, D şi E din R2 (R2.C, R2.D, R2.E).

R1

A	B	C
20	XYZ	30
30	XXZ	20
40	YYX	25

R2

C	D	E
25	XYZ	30
40	YYX	25
30	XXZ	40

Figura 4.2. Două relații unicompatibile

4.3. Operatorii ansamblişti

După cum le spune şi numele, aceşti patru operatori sunt preluaţi din teoria seturilor (mulţimilor). Sunt relativ uşor de înţeles, însă aria lor de folosire este una limitată în practică.

4.3.1. Reuniunea

Reuniunea a două relaţii unicompatibile, R1 şi R2 , este definită astfel: R1 $\cup$ R1 = {tuplu t | t $\in$ R1 sau t $\in$ R1}. Se notează: **R3 ← R1 $\cup$ R2**. Conţinutul tabelei reuniune R3 este prezentat în figura 4.3. Primul tuplu, al treilea şi ultimul din rezultat sunt preluate din R1, iar al doilea şi al patrulea din R2. R3 are numai cinci tupluri deoarece un tuplu este comun tabelelor R1 şi R2. Algebra relaţională elimină automat dublurile (tuplurile identice), astfel încât restricţia de unicitate este asigurată după orice operaţie.

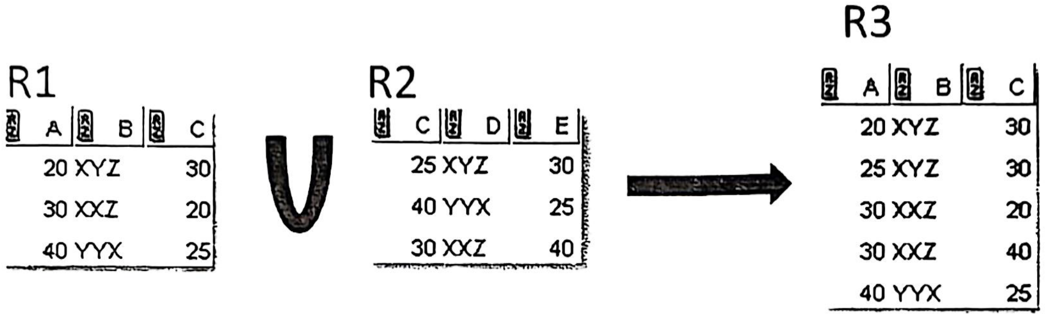


Figura 4.3. Reuniunea a două relaţii

Există suficiente situaţii informaţionale care fac uz de reuniunea a două tabele. Să luăm două exemple:

1. Tabelele ce reflectă tranzacţii economice pot fi descompuse (sparte) în funcţie de anul (luna sau cincinalul, deceniul) la care se referă; dacă ne raportăm la baza noastră de date, tabela LINIIFACT poate fi ruptă în alte două, LF_2000_2001 care ar conţine facturile emise în anii 2000 şi 2001 şi LINIIFACT ce conţine numai înregistrări aferente anului calendaristic 2002. Începând cu 2002, orice situaţie statistică privind vânzările în perioada 2000-2002, 2000-2003 etc. necesită reuniunea celor două tabele, LF_2000_2001 şi LINIIFACT.
2. Pentru a afla care sunt clienţii care au cumpărat cel puţin unul din produsele *Produs 1* şi *Produs 2*, se poate proceda la reuniunea tabelei ce conţine clienţii care au cumpărat *Produs 1* cu tabela clienţilor care au cumpărat *Produs2*.

Reuniunea este comutativă. Singura problemă neclară ar fi legată de numele atributelor în relaţia-rezultat. În acest sens, se poate institui regula potrivit căreia numele atributelor relaţiei-reuniune sunt numele primei relaţii participante în operaţie. Aceasta nu are importanţă asupra comutativităţii, deoarece conţinutul tabelei-rezultat este identic, indiferent care este prima relaţie enumerată.

Ordinea în care apar tuplurile în rezultat nu are nicio importanță, din punctul de vedere al conținutului informațional. În figura 4.3, liniile sunt ordonate după valorile celor trei atribute. Algebra relațională nu garantează în niciun fel maniera de dispunere a tuplurilor în rezultatul unei operațiuni.

4.3.2. Intersecția

Intersecția a două relații unicompatibile, R1 și R2, poate fi definită astfel: R1 ∩ R2 = {tuplu t | t ∈ R1 și t ∈ R2}. Se notează: **R4 ← R1 ∩ R2**. Conținutul tabelei intersecție R4 este prezentat în figura 4.4. Cum numai un tuplu este absolut identic și în R1 și R2, tabela rezultat este alcătuită dintr-o singură linie.

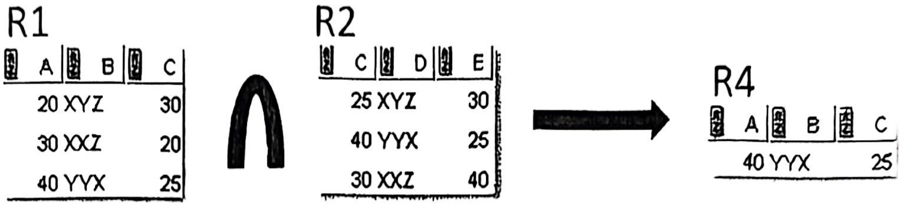


Figura 4.4. Intersecția a două relații

Exemple de informații care fac necesară recurgerea la intersecție:
- pentru a afla care sunt clienții care au cumpărat și *Produs 1*, și *Produs 2*, se poate proceda la intersecția tabelei clienților care au cumpărat *Produs 1* cu tabela alcătuită din clienții care au cumpărat *Produs2;*
- zilele în care s-au făcut vânzări și clientului *Client 1 SRL* și clientului *Client 2 SA;*
- persoanele de la firmele-client care cumulează posturile de *Director vânzări* și *Șef aprovizionare.*

Ca și reuniunea, intersecția este comutativă, iar numele atributelor relației-intersecție sunt extrase din prima relație participantă în operație.

4.3.3. Diferența

Diferența a două relații unicompatibile, notate R1 și R2, este definită astfel: R1 - R2 = {tuplu t | t ∈ R1 și t ∉ R2}. Se notează: **R5 ← R1 - R2**. Conținutul tabelei-diferență R5 (figura 4.5) conține numai tuplurile din prima relație, R1, care nu se regăsesc în a doua relație, R2.

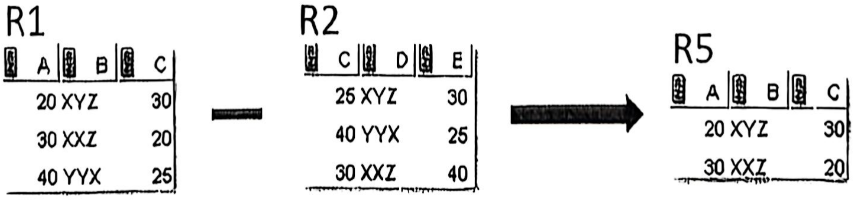


Figura 4.5. Diferența a două relații

Aşadar, din rezultat este eliminat al treilea tuplu din R1, deoarece valorile acestuia există şi în R2 (al doilea tuplu din R2). Exemple ce informaţii care fac necesară recurgerea la diferenţă:

- care sunt clienţii care au cumpărat *Produs 1*, dar nu au cumpărat *Produs 2*?
- care sunt zilele în care s-au făcut vânzări clientului *Client 1 SRL*, dar nu există nicio factură către *Client 2 SA*?

Spre deosebire de reuniune şi intersecţie, diferenţa nu este comutativă. Atributelor relaţiei-diferenţă sunt cele ale primei relaţii (descăzutul), iar tuplurile sunt extrase din relaţia descăzut care nu se regăsesc în relaţia scăzător. În plus, nu există restricţii privind cardinalitatea (numărul de tupluri) celor două relaţii, adică nu este musai ca relaţia descăzut să conţină mai multe tupluri decât cea scăzător.

4.3.4. Produsul cartezian

Produsul cartezian al două relaţii R1 şi R2, denumit de Codd joncţiune încrucişată (CROSS JOIN), este ansamblul tuturor tuplurilor obţinute prin concatenarea fiecărei linii din tabela R1 cu toate liniile tabelei R2. Formal, dacă notăm cele două relaţii: R1 (A1, A2, ..., An) şi R2 (B1, B2, ..., Bm), produsul cartezian este definit astfel: $R1 \otimes R2 = \{(t1\ ,\ t2)\ |\ t1 \in R1 \text{ şi } t2 \in R2\}$. Spre deosebire de celelalte trei operaţiuni precedente, produsul cartezian nu face apel la noţiunea de relaţii uni-compatibile, iar relaţia-rezultat cumulează atributele celor două relaţii-argument. În figura 4.6 este ilustrat rezultatul produsului cartezian al tabelelor R1 şi R2.

R1

A	B	C
20	XYZ	30
30	XXZ	20
40	YYX	25

⊗

R2

C	D	E
25	XYZ	30
40	YYX	25
30	XXZ	40

R6

A	B	R1.C	R2.C	D	E
20	XYZ	30	25	XYZ	30
20	XYZ	30	40	YYX	25
20	XYZ	30	30	XXZ	40
30	XXZ	20	25	XYZ	30
30	XXZ	20	40	YYX	25
30	XXZ	20	30	XXZ	40
40	YYX	25	25	XYZ	30
40	YYX	25	40	YYX	25
40	YYX	25	30	XXZ	40

Figura 4.6. Produsul cartezian

Se notează: **R6 ← R1 ⊗ R2**. Tabela-rezultat R6 are o nouă structură - şase atribute (trei preluate din R1 şi trei din R2). Întrucât există un atribut cu nume comun, C, pentru a diferenţia cele două apariţii, acestea sunt prefixate, în antetul tabelei, cu numele relaţiei din care provine. Prima linie din R6 este obţinută prin „lipirea" primului tuplu din R1 cu primul tuplu din R2, a doua din primul tuplu

din R1 cu al doilea din R2 etc. Cum R1 are trei tupluri, iar R2 tot trei, relaţia--rezultat a produsului cartezian are 3 * 3 = 9 tupluri.

Nu prea există situaţii care să reclame folosirea directă şi exclusivă a produsului cartezian. Cel mai important „merit" al acestuia în algebra relaţională este că permite „alipirea" a două relaţii, fundamentând astfel operatorul-cheie care este joncţiunea.

4.4. Operatorii relaţionali

Cei patru operatori prezentaţi în paragraful precedent sunt generali, spre deosebire de următorii care sunt specifici algebrei relaţionale. De obicei, gruparea operatorilor relaţionali se face astfel: a) *operatori unari de restricţie*, care permit decupajul unei relaţii, pe orizontală - SELECŢIA - şi pe verticală - PROIECŢIA; b) *operatori binari de extensie*: JONCŢIUNEA şi DIVIZIUNEA. O altă deosebire majoră faţă de paragraful precedent este că vom putea recurge şi la exemple concrete din baza de date „cobai" prezentată în capitolul anterior.

4.4.1. Selecţia

Selecţia triază dintr-o relaţie (tabelă) numai tuplurile ce satisfac o condiţie specificată printr-un predicat. Ca preambul, definim noţiunea de *formulă F asupra unei relaţii R*, ca o expresie logică compusă din: *operanzi* care sunt nume de atribute sau constante; *operatori de comparaţie aritmetică*: >, ≥, <, ≤, =, ≠; *operatori logici*: ŞI, SAU, NON. Selecţia unei relaţii R, printr-o condiţie F, notată SF(R), poate fi definită: SF(R) = {tuplu t | (t ∈ R şi F(t) = adevărată)}. O notaţie ceva mai pământeană este: **R1 ← SELECŢIE (R; <expresie-logică>)**, dar, de dragul ştiinţei, vom detalia:

- R este relaţia R (A1, A2, ..., An) asupra căreia se aplică selecţia (Ai sunt atributele sale);
- R1 este noua relaţie obţinută în urma selecţiei, care va avea aceeaşi schemă relaţională cu R - R1 (A1, A2, ..., An);
- <expresie-logică> poate fi scrisă mai analitic astfel:
  - <expresie-logică> = (termen1) şi/sau (termen2) ... şi/sau (termenk), unde
    - termen j = $expresie_1$ ∅ $expresie_2$,
    - $expresie_1$ sau $expresie_2$ sunt expresii calculate plecând de la atributele A_i,
    - ∅ poate fi unul dintre operatorii pentru comparaţie.

Exemplul 1. Pentru a pune în operă savanta notaţie de mai sus, luăm în discuţie o primă problemă: *Care sunt liniile din R1 pentru care valorile atributelor A şi C sunt mai mari decât 20?* Ajungem, astfel, la notaţia:
R ← SELECŢIE (R1; A > 20 AND C > 20).

Tabela R este prezentată în figura 4.7.

A	B	C
40	YYX	25

Figura 4.7. Rezultat selecţie – exemplul 1

Exemplul 2. Care sunt judeţele din Moldova?

Mai întâi, se identifică în baza de date tabela (sau tabelele) din care se extrage rezultatul. În acest exemplu, aceasta este JUDETE. Apoi se stabilesc atributele (atributul) asupra cărora se va aplica predicatul de selecţie. Se obţine:
R ← SELECŢIE (JUDETE; Regiune = 'Moldova') - vezi figura 4.8.

JUD	JUDET	REGIUNE
IS	Iasi	Moldova
VN	Vrancea	Moldova
NT	Neamt	Moldova
SV	Suceava	Moldova
VS	Vaslui	Moldova

Figura 4.8. Rezultat selecţie – exemplul 2

Exemplul 3. Care sunt facturile emise în perioada 2-5 august 2000?

Tabela în care va opera operatorul de selecţie este FACTURI. Predicatul de selecţie utilizează atributul DataFact:

R ← SELECŢIE (FACTURI; DataFact >= '2007-08-02' AND DataFact <= '2007-08-05'.

NRFACT	DATAFACT	CODCL	OBS
1115	02-08-2007	1001	(null)
1116	02-08-2007	1007	Pretul propus initial a fost modificat
1117	03-08-2007	1001	(null)
1118	04-08-2007	1001	(null)

Figura 4.9. Rezultat selecţie – exemplul 3

4.4.2. Proiecţia

Prin proiecţie, o relaţie poate fi „decupată" pe verticală. Dacă selecţia extrage dintr-o tabelă anumite linii, pe baza condiţiei îndeplinite de valorile unora dintre atribute, *proiecţia* permite selectarea într-o tabelă-rezultat numai a coloanelor (atributelor) dorite dintr-o relaţie. Formal, fie o relaţie R (A_1,A_2,...A_n).

Proiecţia relaţiei R asupra unui subansamblu alcătuit din atribute proprii este o relaţie care se obţine după parcurgerea a doi paşi:

- eliminarea dintre A_i a acelor atribute care nu sunt specificate;
- suprimarea dublurilor (tuplurile identice).

Se notează: **R1 ← PROIECȚIE (R; Aj, Ak, ..., Ax)**. Spre deosebire de R, schema relației R_1 este alcătuită numai din atributele indicate: R1 (A_j, A_k, ..., A_x). Dacă după extragerea coloanelor nu există tupluri (linii) identice, R1 va avea același număr de linii ca și relația R. În caz contrar, numărul lor va fi mai mic, în funcție de numărul dublurilor.

Exemplul 4. Începem, ca de obicei, cu un exemplu ceva mai arid. Care sunt valorile combinației atributelor A și C în relația R1?

R ← PROIECȚIE (R1; A, C).

Tabela R are două coloane, A și C, și trei linii, ca în figura 4.10.

A	C
20	30
30	20
40	25

Figura 4.10. Rezultat proiecție – exemplul 4

Exemplul 5. Ce regiuni ale țării sunt preluate în bază?

Tabela în care se află răspunsul este JUDETE. Singura coloană care interesează este Regiune:

R ← PROIECȚIE (JUDETE; Regiune).

În primul pas, se face decupajul pe verticală, obținându-se o relația notată R′, apoi se elimină dublurile, rezultatul final fiind relația R.

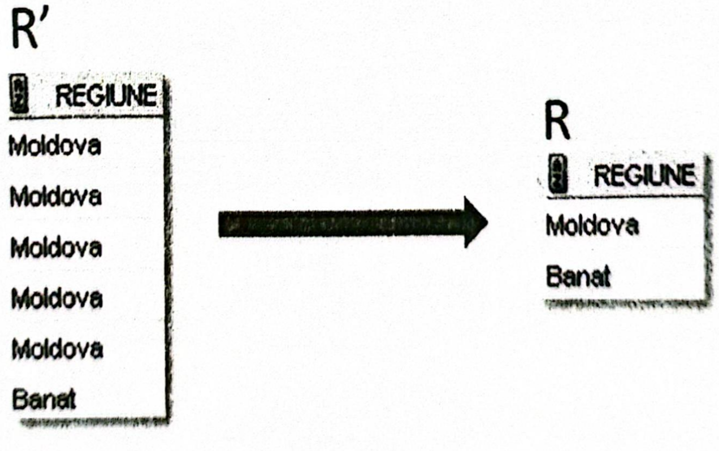


Figura 4.11. Rezultat proiecție – exemplul 5

Exemplul 6. Care sunt: codul, denumirea și numărul de telefon ale fiecărui client?

Tabela care interesează este CLIENȚI, din care se decupează trei coloane - CodCl, DenCl și Telefon.

R ← PROIECȚIE (CLIENTI; CodCl, DenCl, Telefon).

CODCL	DENCL	TELEFON
1001	Client 1 SRL	(null)
1002	Client 2 SA	032-212121
1003	Client 3 SRL	035-222222
1004	Client 4	(null)
1005	Client 5 SRL	056-111111
1006	Client 6 SA	(null)
1007	Client 7 SRL	056-121212

Figura 4.12. Rezultat proiecţie – exemplul 6

4.4.3. Înlănţuirea consultărilor

Rezultatul unei consultări este o relaţie (tabelă) nouă. Pe baza acestui fapt, se pot înlănţui două sau mai multe operaţiuni, redactându-se astfel interogări complexe.

Exemplul 7. Care este numărul de telefon al clientul Client 2 SA?

Soluţia este una foarte simpă. Cu ajutorul selecţiei se decupează din relaţia CLIENŢI numai linia corespunzăoare clientului „incriminat". Se obţine o relaţie nou-nouţă denumită (de noi) R′. Asupra lui R′ se aplică o proiecţie, deoarece ne interesează numai numărul de telefon; astfel, R conţine răspunsul la problema luată în discuţie (vezi figura 4.13).

R′ ← SELECŢIE (CLIENTI; DenCl = 'Client 2 SA')
R ← PROIECŢIE (R′; Telefon)

R′

CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1002	Client 2 SA	R1002	(null)	700505	032-212121

R

TELEFON
032-212121

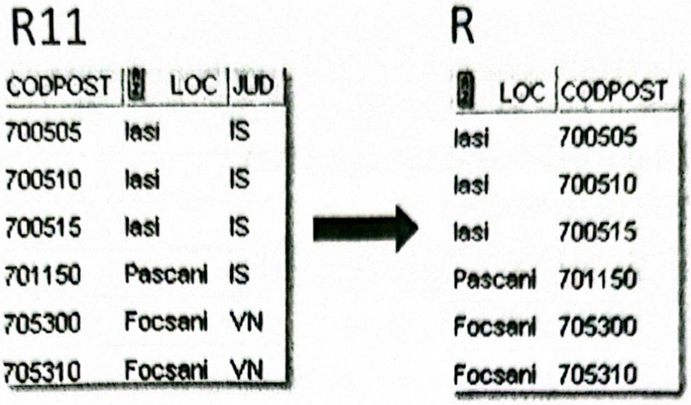
Figura 4.13. Înlănţuirea unei selecţii cu o proiecţie – exemplul 7

Exemplul 8. Care sunt denumirile şi codurile poştale introduse pentru localităţile (prezente în bază) din judeţele Iaşi (IS) şi Vrancea (VN)?

Tabela „interogată" este CODURI_POŞTALE. Pentru a răspunde la întrebarea pe care tot noi am formulat-o, putem alege între următoarele două soluţii:

Soluţia 1 – figura 4.14:

R11 ← SELECŢIE (CODURI_POSTALE; Jud = 'IS' OR Jud = 'VN')
R ← PROIECŢIE (R11; Loc, CodPost)



R11

CODPOST	LOC	JUD
700505	Iasi	IS
700510	Iasi	IS
700515	Iasi	IS
701150	Pascani	IS
705300	Focsani	VN
705310	Focsani	VN

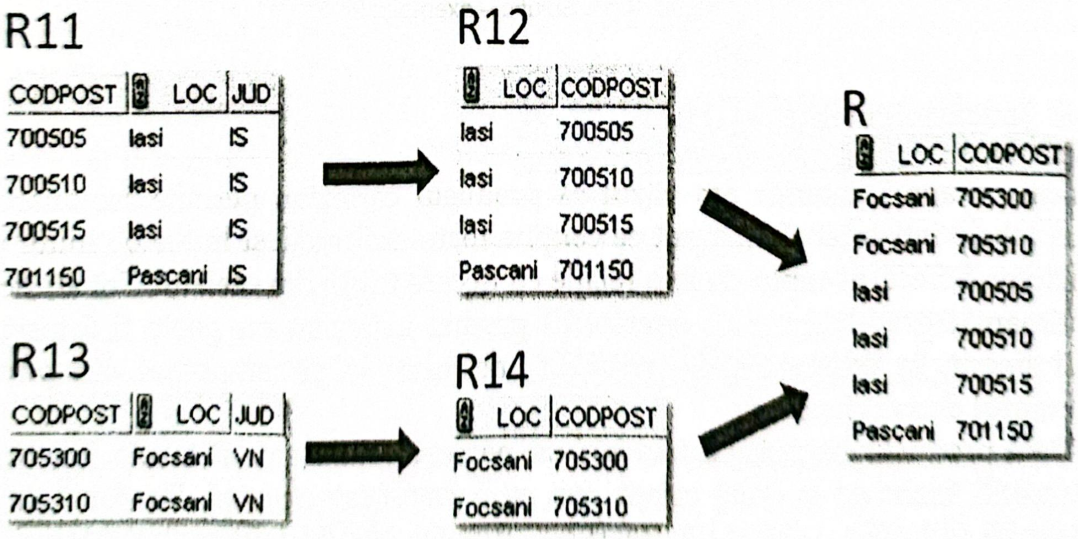
R

LOC	CODPOST
Iasi	700505
Iasi	700510
Iasi	700515
Pascani	701150
Focsani	705300
Focsani	705310

Figura 4.14. Exemplu 8 – soluția 1

Soluția 2 - figura 4.15:

R11 ← SELECȚIE (CODURI_POSTALE; Jud = 'IS')
R12 ← PROIECȚIE (R11; Loc, CodPost)
R13 ← SELECȚIE (CODURI_POSTALE; Jud = 'VN')
R14 ← PROIECȚIE (R13; Loc, CodPost)
R ← R12 ∪ R14



R11

CODPOST	LOC	JUD
700505	Iasi	IS
700510	Iasi	IS
700515	Iasi	IS
701150	Pascani	IS

R12

LOC	CODPOST
Iasi	700505
Iasi	700510
Iasi	700515
Pascani	701150

R13

CODPOST	LOC	JUD
705300	Focsani	VN
705310	Focsani	VN

R14

LOC	CODPOST
Focsani	705300
Focsani	705310

R

LOC	CODPOST
Focsani	705300
Focsani	705310
Iasi	700505
Iasi	700510
Iasi	700515
Pascani	701150

Figura 4.15. Exemplul 8 – soluția 2

Prima soluție este, prin simplitate, cea mai tentantă. Este, însă, un prim caz în care pentru rezolvarea unei probleme pot fi formulate două (sau mai multe) soluții. Îmi fac (cu plăcere) datoria de a vă bate la cap că, deși ordinea liniilor este diferită, rezultatele finale (tabelele R) din figurile 4.14 și 4.15 sunt identice din punctul de vedere al conținutului informațional.

Exemplul 9. Care sunt codurile produselor care apar deopotrivă în factura 1111 și în factura 1117?

Tabela din care vor fi extrase datele este LINIIFACT. Soluţia se bazează pe intersecţia relaţiei care conţine produsele prezente în factura 1111 (R2) cu relaţia produselor prezente în factura 1117 (R4), după cum reiese din figura 4.16.

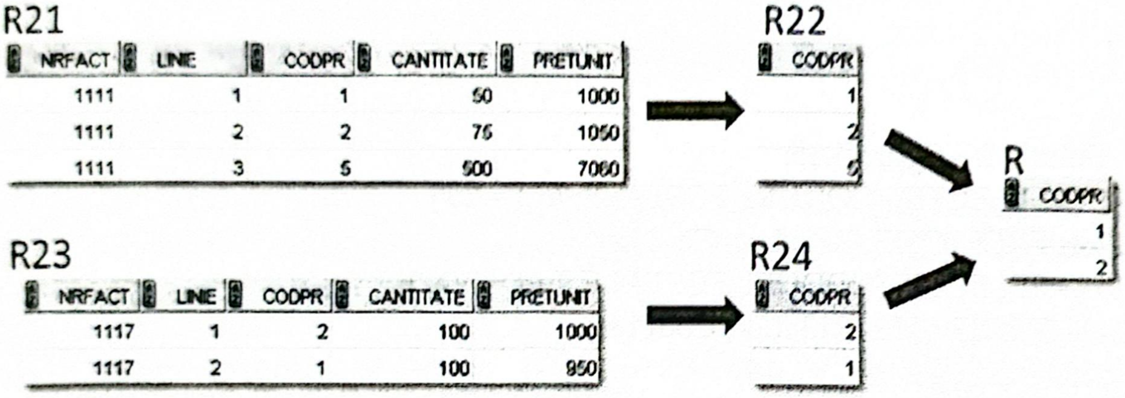
R21 ← SELECŢIE (LINIIFACT; NrFact = 1111)

R22 ← PROIECŢIE (R21; CodPr)

R23 ← SELECŢIE (LINIIFACT; NrFact = 1117)

R24 ← PROIECŢIE (R23; CodPr)

R ← R22 ∩ R24



R21

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1111	1	1	50	1000
1111	2	2	75	1050
1111	3	5	500	7000

R22

CODPR
1
2
5

R23

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1117	1	2	100	1000
1117	2	1	100	950

R24

CODPR
2
1

R

CODPR
1
2

Figura 4.16. Soluţie – exemplul 9

4.4.4. Joncţiuni interne

Într-un paragraf anterior am văzut că produsul cartezian permite fuzionarea a două tabele într-o tabelă-mamut ce conţine toate atributele şi liniile obţinute prin combinarea fiecărui tuplu dintr-o relaţie cu fiecare tuplu din cealaltă. Tot atunci ne exprimam regretul sincer că operatorul produs cartezian nu poate fi folosit, de unul singur, în interogări, dar că acest „neajuns" va fi compensat din plin de operatorul derivat *joncţiune*.

Dacă produsul cartezian este o fuziune necondiţionată a două tabele, joncţiunea reprezintă fuziunea a două relaţii *care au o proprietate comună*. Fie două relaţii, notate: R1 ($A1, A2, ..., An$) şi R2 ($B1, B2, ..., Bp$). Fie Ai şi Bj două atribute definite pe acelaşi domeniu şi θ ansamblul operatorilor de comparaţie: $\{=, >, \geq, <, \leq, \neq\}$, ce pot fi aplicaţi celor două atribute Ai şi Bj. Joncţiunea relaţiei R1, prin Ai, cu relaţia R2, prin Bj, notată R1 (Ai θ Bj) R2 sau R1 $\bowtie_{\theta}$ R2, este relaţia ale cărei tupluri sunt obţinute prin concatenarea fiecărui tuplu al relaţiei R1 cu tuplurile relaţiei R2, pentru care este verificată condiţia θ instituită între Ai şi Bj. R1 (Ai θ Bj) R2 = $\{t \mid t \in R1 \otimes R2$ şi $t(Ai)\ \theta\ t(Bj)\}$.

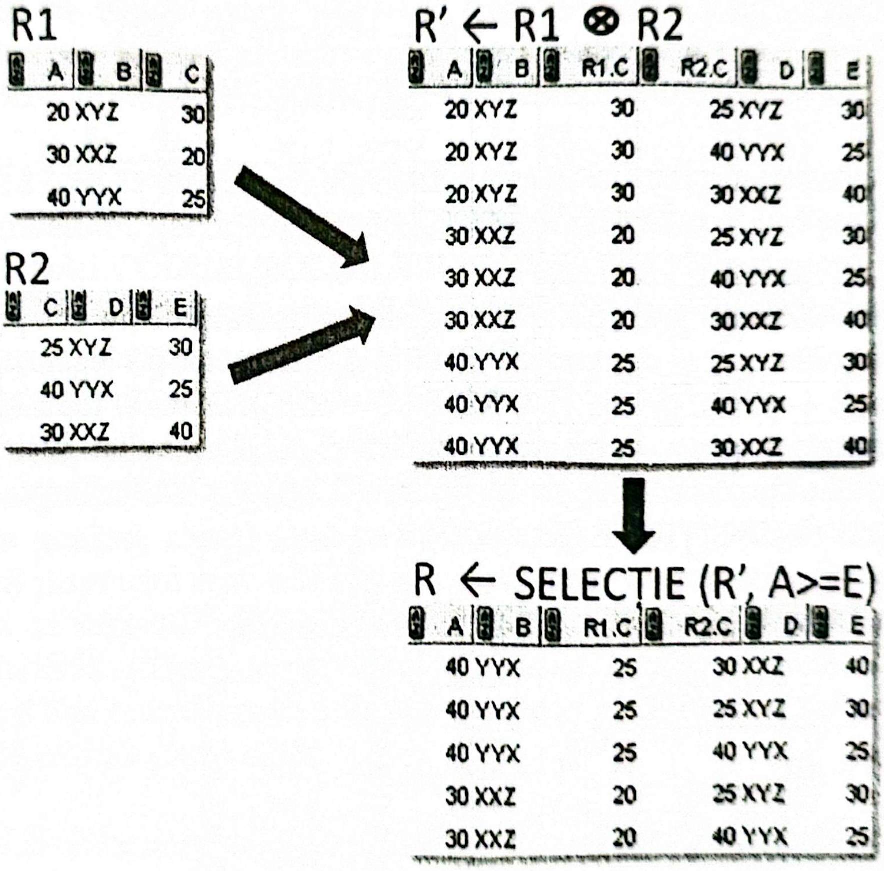
Joncţiunea internă este echivalentă unui produs cartezian urmat de o selecţie. Joncţiunea internă definită mai sus este referită în lucrările de specialitate ca *theta-joncţiune.* În lucrul cu BDR se utilizează cu precădere *echijoncţiunea,* ce reprezentă un caz particular al theta-joncţiunii, atunci când θ este operatorul de egalitate („="). Formal, echijoncţiunea se defineşte astfel: R1 ($Ai = Bj$), R2 = $\{t \mid t \in R1 \otimes R2$

şi t(A*i*) = t(B*j*)}. Apelăm şi la o altă notaţie, mai uşor de reprezentat şi suficient de inteligibilă.

R ← ECHIJONCȚIUNE (R1, R2; Ai=Bj)

Exemplul 10. Theta-joncţiune

Începem exemplificările cu aceleaşi două tabele folosite din paragraful precedent, R1 şi R2. Rezultatul joncţiunii (theta-joncţiunii) exprimată prin expresia: **R ← JONCŢIUNE (R1, R2; R1.A >= R2.E)** va fi obţinut în doi paşi, după cum este descris în figura 4.17.



R1

A	B	C
20	XYZ	30
30	XXZ	20
40	YYX	25

R2

C	D	E
25	XYZ	30
40	YYX	25
30	XXZ	40

R' ← R1 ⊗ R2

A	B	R1.C	R2.C	D	E
20	XYZ	30	25	XYZ	30
20	XYZ	30	40	YYX	25
20	XYZ	30	30	XXZ	40
30	XXZ	20	25	XYZ	30
30	XXZ	20	40	YYX	25
30	XXZ	20	30	XXZ	40
40	YYX	25	25	XYZ	30
40	YYX	25	40	YYX	25
40	YYX	25	30	XXZ	40

R ← SELECȚIE (R', A>=E)

A	B	R1.C	R2.C	D	E
40	YYX	25	30	XXZ	40
40	YYX	25	25	XYZ	30
40	YYX	25	40	YYX	25
30	XXZ	20	25	XYZ	30
30	XXZ	20	40	YYX	25

Figura 4.17. Mecanismul de theta-joncționare – exemplul 10

Exemplul 11. Echijoncţiune

Operatorul de comparaţie dintre cele două atribute este, obligatoriu, semnul de egalitate.

R ← JONCŢIUNE (R1, R2; R1.A = R2.E)

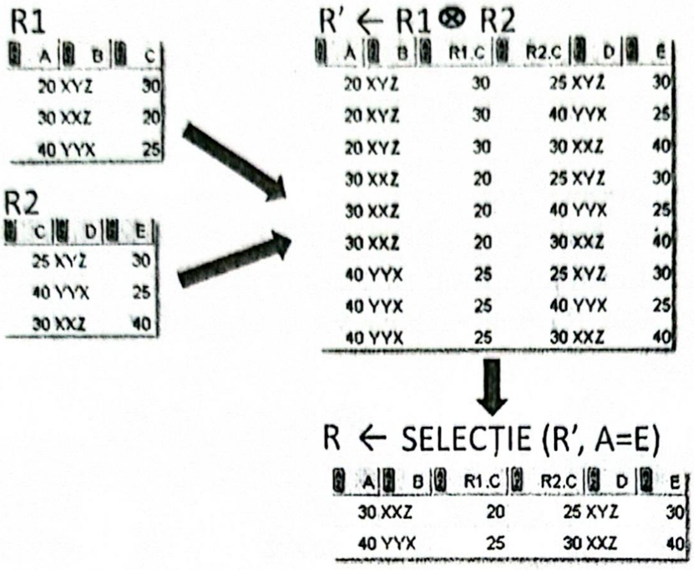


Figura 4.18. Echijoncțiune – exemplul 11

Exemplul 12. Joncțiune naturală

Joncțiunea naturală presupune nu numai ca operatorul de comparație să fie semnul de egalitate, ci și denumirea identică a atributelor de legătură dintre cele două tabele. R ← JONCȚIUNE (R1, R2; R1.C = R2.C). Datorită faptului că ambele atribute au același nume, se poate considera că tabela-rezultat păstrează numai unul dintre cele două atribute, ca în figura 4.19 și se poate recurge la o notație simplificată: R ← JONCȚIUNE (R1, R2; C).

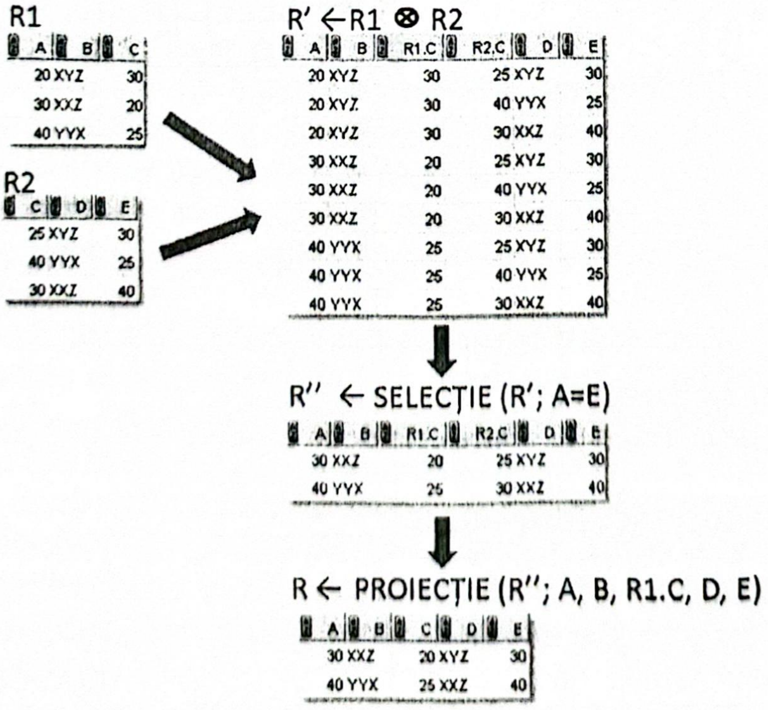


Figura 4.19. Joncțiune naturală – exemplul 12

De fapt, joncțiunea naturală poate fi privită ca o proiecție pe echijoncțiune, din rezultatul echijoncțiunii selectându-se toate atributele, cu excepția unui „rând" din atributele după care se face joncționarea.

De ce se insistă atât de mult pe importanța joncțiunii? În primul rând pentru că permite recompunerea relației universale inițiale. Modelul relațional se bazează pe spargerea bazei în relații, astfel încât nivelul redundanței datelor și problemele la actualizarea tabelelor să fie reduse la minim. Cele mai multe interogări, însă, operează cu date și predicate aplicate simultan atributelor din două sau mai multe tabele. Cum selecția este un operator *unar* (poate fi aplicată unei singure relații), este necesară fuzionarea prealabilă a celor două, trei... relații și obținerea unei relații-agregat, relație-agregat la care se aplică predicatul suplimentar de selecție.

Fuzionarea este posibilă prin joncțiune. Prin joncționarea tuturor relațiilor dintr-o bază de date se obține relația *universală* (cea inițială, atotcuprinzătoare). Nu ne permitem să reconstituim structura și conținutul relației universale ale bazei de date VÂNZARI, însă, prin exemple, vom încercă să demonstrăm utilitatea joncțiunii.

Exemplul 13. Să se obțină, pentru fiecare localitate, codurile poștale, denumirea, indicativul județului, denumirea județului și regiunea din care face parte.

Practic, tabelei CODURI_POSTALE îi trebuie „alipite" la dreapta informațiile din tabela JUDEȚE. Problema este dificil de formulat, însă rezolvarea sa este cât se poate de simplă – joncționarea celor două relații:

R ← JONCȚIUNE (CODURI_POSTALE, JUDETE; Jud)

Una dintre întrebările „clasice" pentru verificarea modului în care a fost sau nu înțeleasă joncțiunea internă este: *Câte linii are tabela-rezultat a joncțiunii*? În cazul nostru și, în general, atunci când se joncționează două tabele aflate în posturile părinte-copil răspunsul este: *câte linii are tabela-copil*. Răspunsul este corect numai atunci când se respectă integritatea referențială, altfel spus, numai atunci când toate valorile cheii străine sunt nenule și se regăsesc în tabela-părinte. Ca de obicei, lucrurile sunt mai complicate în realitate, deoarece joncțiunea nu se instituie musai între o tabelă-părinte și una copil.

Exemplul 14. Care sunt codurile poștale ale localităților din Banat?

Soluția 1 – după calapodul exemplului anterior:

R11 ← JONCȚIUNE (CODURI_POȘTALE, JUDEȚE; Jud)

R ← SELECȚIE (R11; Regiune = 'Banat')

Soluția 2 – mai întâi, se aplică selecția asupra tabelei JUDEȚE, iar tabela intermediară se joncționează cu CODURI_POȘTALE:

R12 ← SELECȚIE (JUDEȚE; Regiune = 'Banat')

R ← JONCȚIUNE (CODURI_POȘTALE, R12; Jud)

A doua variantă pare mai bună decât prima, deoarece joncțiunea operează asupra a două relații mai reduse ca dimensiuni (vezi figura 4.20). Diferența este cu atât mai vizibilă atunci când relația JUDEȚE conține toate județele țării, iar CODURI_POȘTALE are câteva sute de înregistrări. Iar dacă ne gândim că înaintea oricărei joncțiuni se calculează produsul cartezian, apare drept firească ideea amânării joncțiunii, astfel încât aceasta să opereze asupra unor tabele cu un număr cât mai mic de linii și coloane.

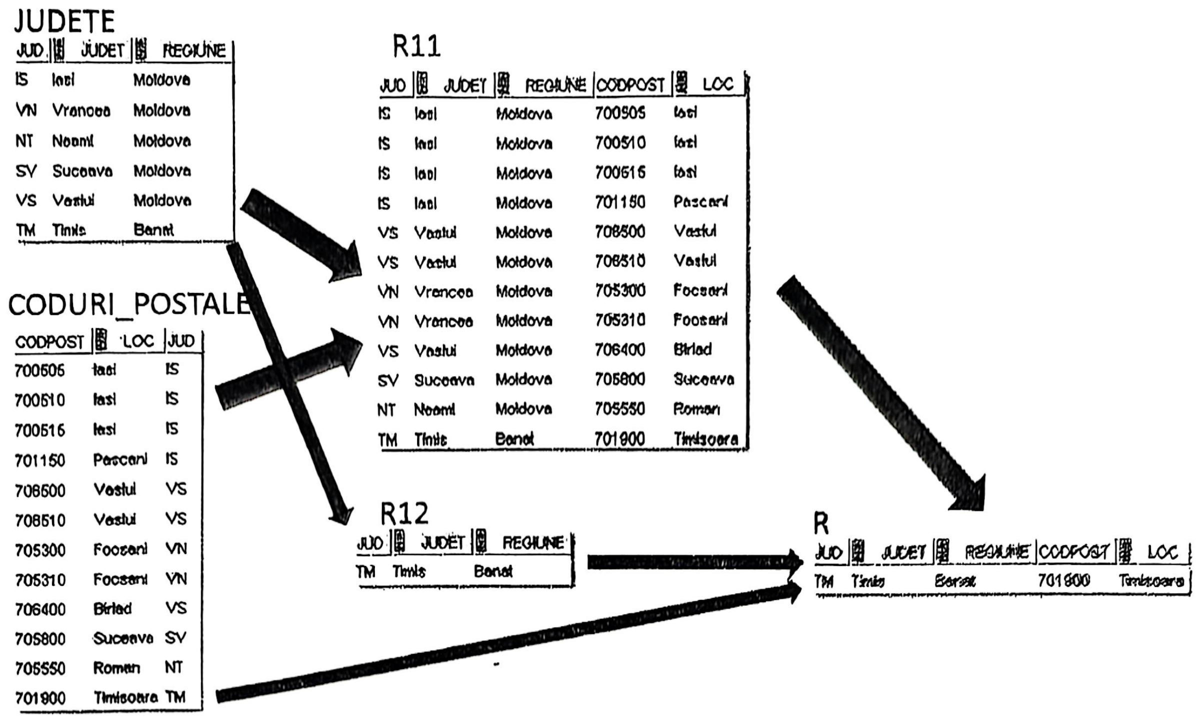


Figura 4.20. Ilustrarea modului de execuţie a celor două soluţii AR – exemplul 14

Coborând cu picioarele pe pământ, trebuie spus că discuţia noastră are un caracter de principiu, deoarece algebra relaţională este, totuşi, un limbaj... curat teoretic.

Exemplul 15. În ce zile s-a vândut produsul cu denumirea „Produs 1"?

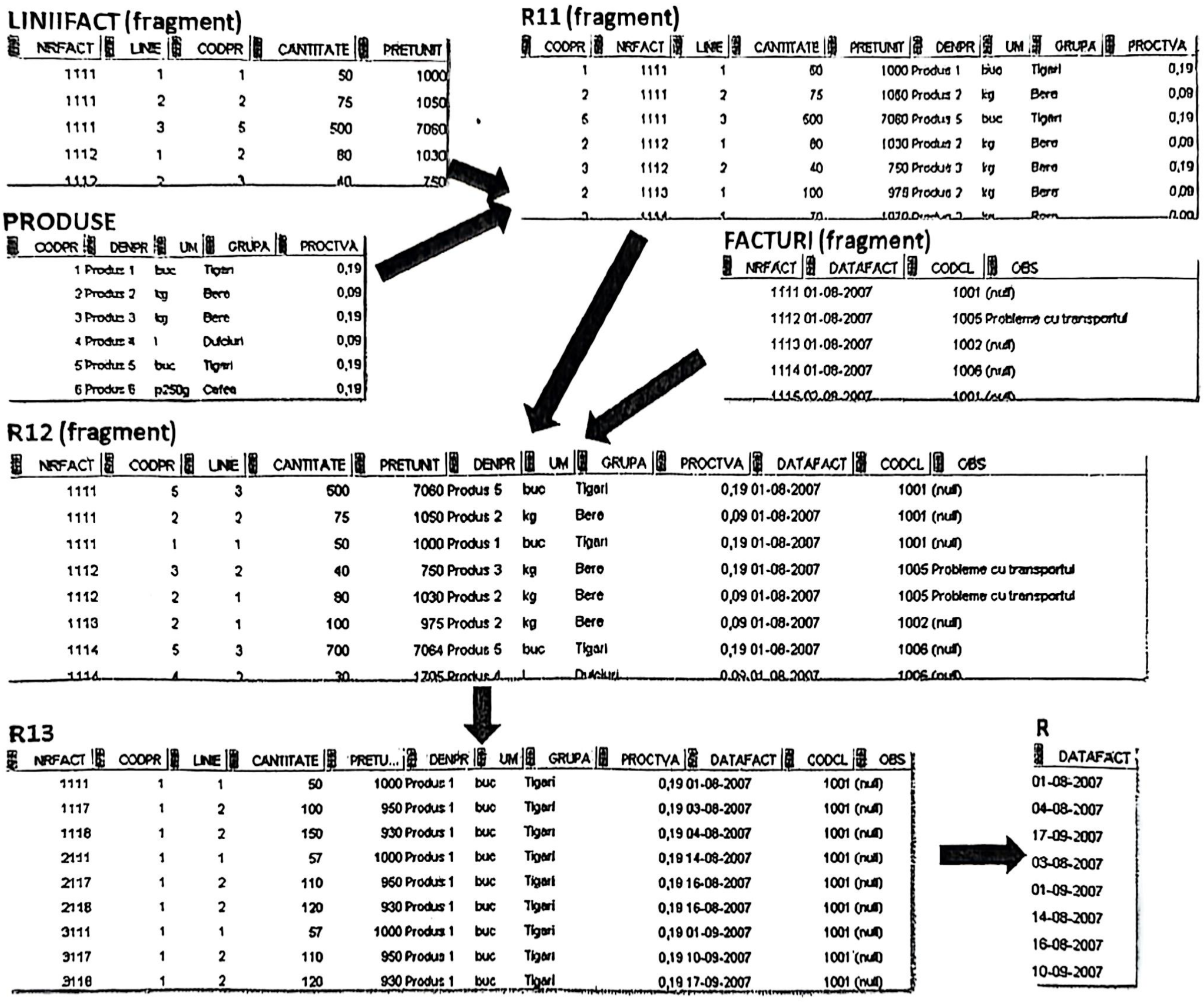
Elementul de noutate îl reprezintă interesul pentru o informaţie ce provine dintr-o relaţie (atributul **DataFact** din FACTURI) pe baza unei condiţii aplicate altei relaţii (atributul **DenPr** din PRODUSE), iar cele două relaţii, FACTURI şi PRODUSE nu sunt în raportul părinte-copil. În aceste cazuri, este necesară atragerea altor relaţii, până se completează „lanţul". Interogarea ce rezolvă problema ridicată în acest exemplu necesită şi tabela LINIIFACT.

Soluţia 1 - neoptimizată:

R11 ← JONCŢIUNE (PRODUSE, LINIIFACT; CodPr)
R12 ← JONCŢIUNE (R11, FACTURI; NrFact)
R13 ← SELECŢIE (R12; DenPr = 'Produs 1')
R ← PROIECŢIE (R13; DataFact)

Soluţia 2 - optimizată:

R11 ← SELECŢIE (PRODUSE; DenPr = 'Produs 1')
R12 ← PROIECŢIE (R11; CodPr)
R13 ← JONCŢIUNE (R12, LINIIFACT; CodPr)
R14 ← PROIECŢIE (R13; NrFact)
R15 ← JONCŢIUNE (R14, FACTURI; NrFact)
R ← PROIECŢIE (R15; DataFact)



LINIIFACT (fragment)

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1111	1	1	50	1000
1111	2	2	75	1050
1111	3	5	500	7060
1112	1	2	80	1030

PRODUSE

CODPR	DENPR	UM	GRUPA	PROCTVA
1	Produs 1	buc	Tigari	0,19
2	Produs 2	kg	Bere	0,09
3	Produs 3	kg	Bere	0,19
4	Produs 4	l	Dulciuri	0,09
5	Produs 5	buc	Tigari	0,19
6	Produs 6	p250g	Cafea	0,19

R11 (fragment)

CODPR	NRFACT	LINIE	CANTITATE	PRETUNIT	DENPR	UM	GRUPA	PROCTVA
1	1111	1	50	1000	Produs 1	buc	Tigari	0,19
2	1111	2	75	1050	Produs 2	kg	Bere	0,09
5	1111	3	500	7060	Produs 5	buc	Tigari	0,19
2	1112	1	80	1030	Produs 2	kg	Bere	0,09
3	1112	2	40	750	Produs 3	kg	Bere	0,19
2	1113	1	100	975	Produs 2	kg	Bere	0,09

FACTURI (fragment)

NRFACT	DATAFACT	CODCL	OBS
1111	01-08-2007	1001	(null)
1112	01-08-2007	1005	Probleme cu transportul
1113	01-08-2007	1002	(null)
1114	01-08-2007	1006	(null)

R12 (fragment)

NRFACT	CODPR	LINIE	CANTITATE	PRETUNIT	DENPR	UM	GRUPA	PROCTVA	DATAFACT	CODCL	OBS
1111	5	3	500	7060	Produs 5	buc	Tigari	0,19	01-08-2007	1001	(null)
1111	2	2	75	1050	Produs 2	kg	Bere	0,09	01-08-2007	1001	(null)
1111	1	1	50	1000	Produs 1	buc	Tigari	0,19	01-08-2007	1001	(null)
1112	3	2	40	750	Produs 3	kg	Bere	0,19	01-08-2007	1005	Probleme cu transportul
1112	2	1	80	1030	Produs 2	kg	Bere	0,09	01-08-2007	1005	Probleme cu transportul
1113	2	1	100	975	Produs 2	kg	Bere	0,09	01-08-2007	1002	(null)
1114	5	3	700	7064	Produs 5	buc	Tigari	0,19	01-08-2007	1006	(null)

R13

NRFACT	CODPR	LINIE	CANTITATE	PRETU...	DENPR	UM	GRUPA	PROCTVA	DATAFACT	CODCL	OBS
1111	1	1	50	1000	Produs 1	buc	Tigari	0,19	01-08-2007	1001	(null)
1117	1	2	100	950	Produs 1	buc	Tigari	0,19	03-08-2007	1001	(null)
1118	1	2	150	930	Produs 1	buc	Tigari	0,19	04-08-2007	1001	(null)
2111	1	1	57	1000	Produs 1	buc	Tigari	0,19	14-08-2007	1001	(null)
2117	1	2	110	950	Produs 1	buc	Tigari	0,19	16-08-2007	1001	(null)
2118	1	2	120	930	Produs 1	buc	Tigari	0,19	16-08-2007	1001	(null)
3111	1	1	57	1000	Produs 1	buc	Tigari	0,19	01-09-2007	1001	(null)
3117	1	2	110	950	Produs 1	buc	Tigari	0,19	10-09-2007	1001	(null)
3118	1	2	120	930	Produs 1	buc	Tigari	0,19	17-09-2007	1001	(null)

R

DATAFACT
01-08-2007
04-08-2007
17-09-2007
03-08-2007
01-09-2007
14-08-2007
16-08-2007
10-09-2007

Figura 4.21. Ilustrarea modului de execuție a primei soluții AR – exemplul 15

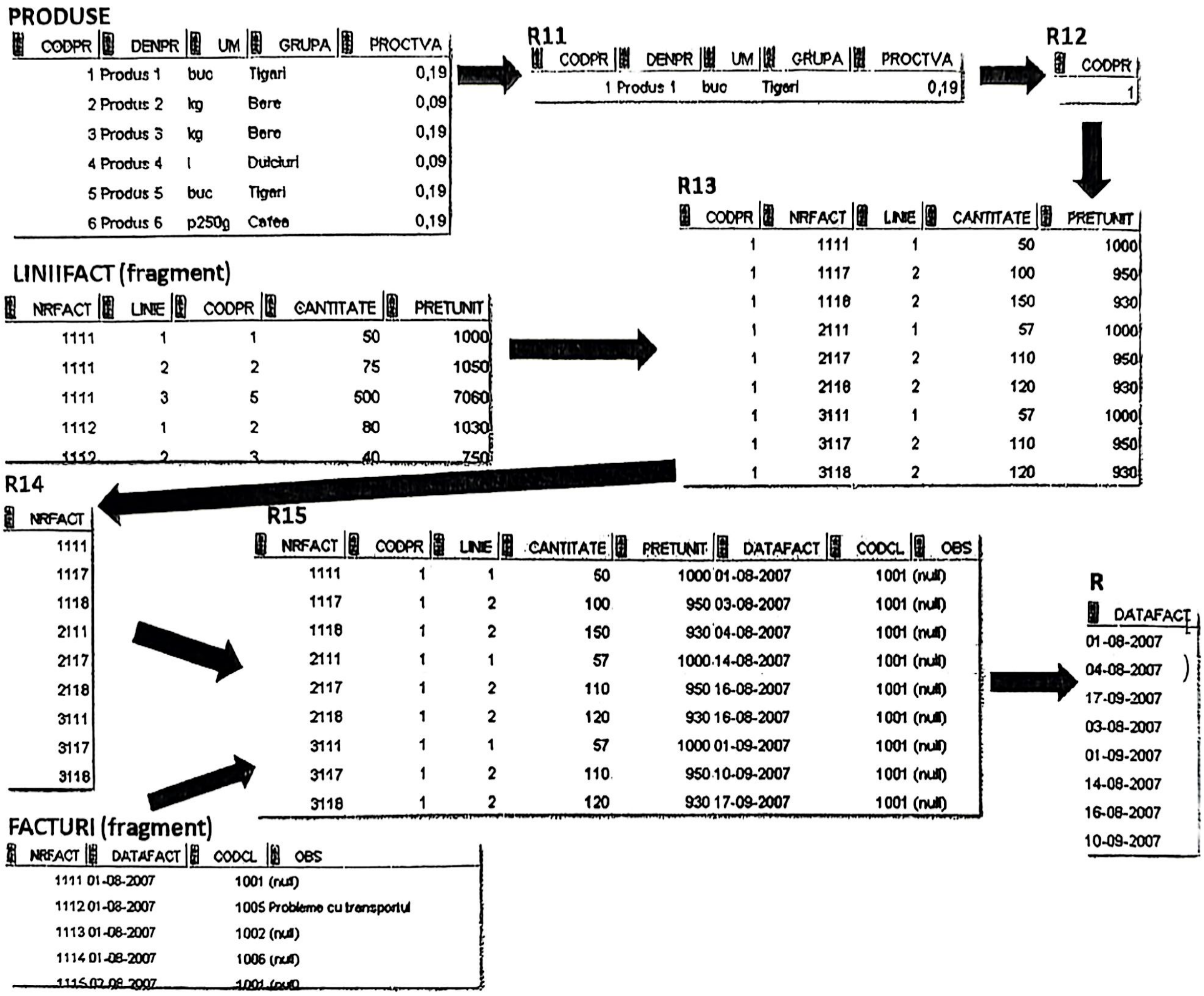
În cea de-a doua soluție (vezi figura 4.22), fideli principiului „joncțiunii celei mai economicoase", am eliminat nu numai tuplurile, ci și atributele de prisos înaintea calculării relației intermediare.

Exemplul 16. În ce județe s-a vândut produsul cu denumirea „Produs 1" în perioada 3-5 august 2007?

Relația-rezultat trebuie să conțină valori ale atributului Judet din tabela JUDEȚE. Predicatul de selecție se aplică, însă, în alte două tabele: PRODUSE, în care DenPr = 'Produs 1', și FACTURI, ale cărei tupluri trebuie să verifice condiția: DataFact >= '2007/08/03' AND DataFact <= '2007/08/05'.

R11 ← SELECȚIE (PRODUSE; DenPr = "Produs 1")
R12 ← JONCȚIUNE (R11, LINIIFACT; CodPr)
R13 ← PROIECȚIE (R12; NrFact)
R14 ← SELECȚIE(FACTURI; DataFact >= '2007/08/03' AND
DataFact <= '2007/08/05')
R5 ← JONCȚIUNE (R3, R4; NrFact)
R6 ← JONCȚIUNE (R5, CLIENȚI; CodCl)
R7 ← PROIECȚIE (R6; CodPost)
R8 ← JONCȚIUNE (R7, CODURI_POȘTALE; CodPost)

R9 ← PROIECȚIE (R8; Jud)
R10 ← JONCȚIUNE (R9, JUDEȚE; Jud)
R ← PROIECȚIE (R10; Judet)



PRODUSE

CODPR	DENPR	UM	GRUPA	PROCTVA
1	Produs 1	buc	Tigari	0,19
2	Produs 2	kg	Bere	0,09
3	Produs 3	kg	Bere	0,19
4	Produs 4	l	Dulciuri	0,09
5	Produs 5	buc	Tigari	0,19
6	Produs 6	p250g	Cafea	0,19

R11

CODPR	DENPR	UM	GRUPA	PROCTVA
1	Produs 1	buc	Tigari	0,19

R12

CODPR
1

LINIIFACT (fragment)

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1111	1	1	50	1000
1111	2	2	75	1050
1111	3	5	500	7060
1112	1	2	80	1030
1112	2	3	40	750

R13

CODPR	NRFACT	LINIE	CANTITATE	PRETUNIT
1	1111	1	50	1000
1	1117	2	100	950
1	1118	2	150	930
1	2111	1	57	1000
1	2117	2	110	950
1	2118	2	120	930
1	3111	1	57	1000
1	3117	2	110	950
1	3118	2	120	930

R14

NRFACT
1111
1117
1118
2111
2117
2118
3111
3117
3118

R15

NRFACT	CODPR	LINIE	CANTITATE	PRETUNIT	DATAFACT	CODCL	OBS
1111	1	1	50	1000	01-08-2007	1001	(null)
1117	1	2	100	950	03-08-2007	1001	(null)
1118	1	2	150	930	04-08-2007	1001	(null)
2111	1	1	57	1000	14-08-2007	1001	(null)
2117	1	2	110	950	16-08-2007	1001	(null)
2118	1	2	120	930	16-08-2007	1001	(null)
3111	1	1	57	1000	01-09-2007	1001	(null)
3117	1	2	110	950	10-09-2007	1001	(null)
3118	1	2	120	930	17-09-2007	1001	(null)

FACTURI (fragment)

NRFACT	DATAFACT	CODCL	OBS
1111	01-08-2007	1001	(null)
1112	01-08-2007	1005	Probleme cu transportul
1113	01-08-2007	1002	(null)
1114	01-08-2007	1006	(null)
1115	02-08-2007	1001	(null)

R

DATAFACT
01-08-2007
04-08-2007
17-09-2007
03-08-2007
01-09-2007
14-08-2007
16-08-2007
10-09-2007

Figura 4.22. Ilustrarea modului de execuție a celei de-a doua soluții AR – exemplul 15

Exemplul 17. În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"?

Rezultatul conține atributul **DataFact**. Predicatul de selecție se aplică tabelei PRODUSE.

Soluția 1:

R11 ← SELECȚIE (PRODUSE; DenPr = 'Produs 1')
R12 ← JONCȚIUNE (R11, LINIIFACT; CodPr)
R13 ← JONCȚIUNE (R12, FACTURI; NrFact)
R14 ← PROIECȚIE (R13; DataFact)
R15 ← SELECȚIE (PRODUSE; DenPr = 'Produs 2')
R16 ← JONCȚIUNE (R15, LINIIFACT; CodPr)
R17 ← JONCȚIUNE (R16, FACTURI; NrFact)
R18 ← PROIECȚIE (R17; DataFact)
R ← R14 ∩ R8

Soluția 2:
R21 ← SELECȚIE (PRODUSE; DenPr = 'Produs 1')
R22 ← JONCȚIUNE (R21, LINIIFACT; CodPr)
R23 ← JONCȚIUNE (R22, FACTURI; NrFact)
R24 ← SELECȚIE (PRODUSE; DenPr = 'Produs 2')
R25 ← JONCȚIUNE (R25, LINIIFACT; CodPr)
R26 ← JONCȚIUNE (R26, FACTURI; NrFact)
R27 ← JONCȚIUNE (R23, R26; DataFact)
R ← PROIECȚIE (R27; DataFact)

Prima variantă este una „cuminte". Se intersectează relația zilelor în care s-a vândut primul produs (R14) cu relația zilelor în care s-a facturat produsul 2 (R18). A doua e ceva mai insolită. Pur și simplu, în loc de intersecție folosim joncțiunea. Cum, spre deosebire de intersecție, relațiile joncționate nu trebuie să fie unicompatibile, putem să operăm direct joncțiunea între R23 și R26.

Reținem ideea: putem să simulăm intersecția a două relații prin joncțiune. Pentru a fi mai convingător, rogu-vă să examinați figura 4.23 în care, pornind de la relațiile R1 și R2, se obține relația-intersecție R direct prin operatorul intersecție (stânga figurii) și mai pe ocolite, folosind joncțiunea (dreapta).

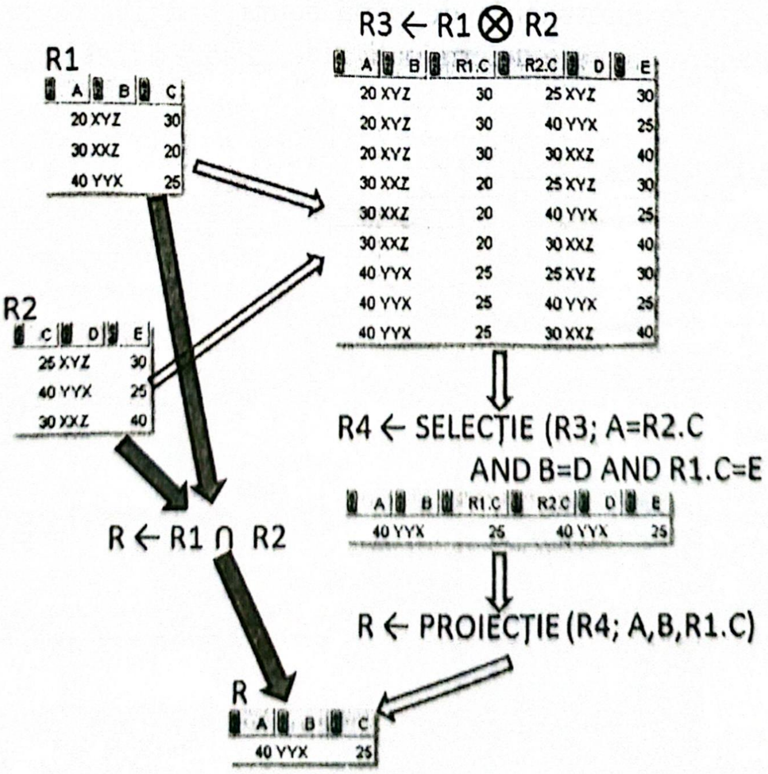


Figura 4.23. Intersecția prin joncțiune

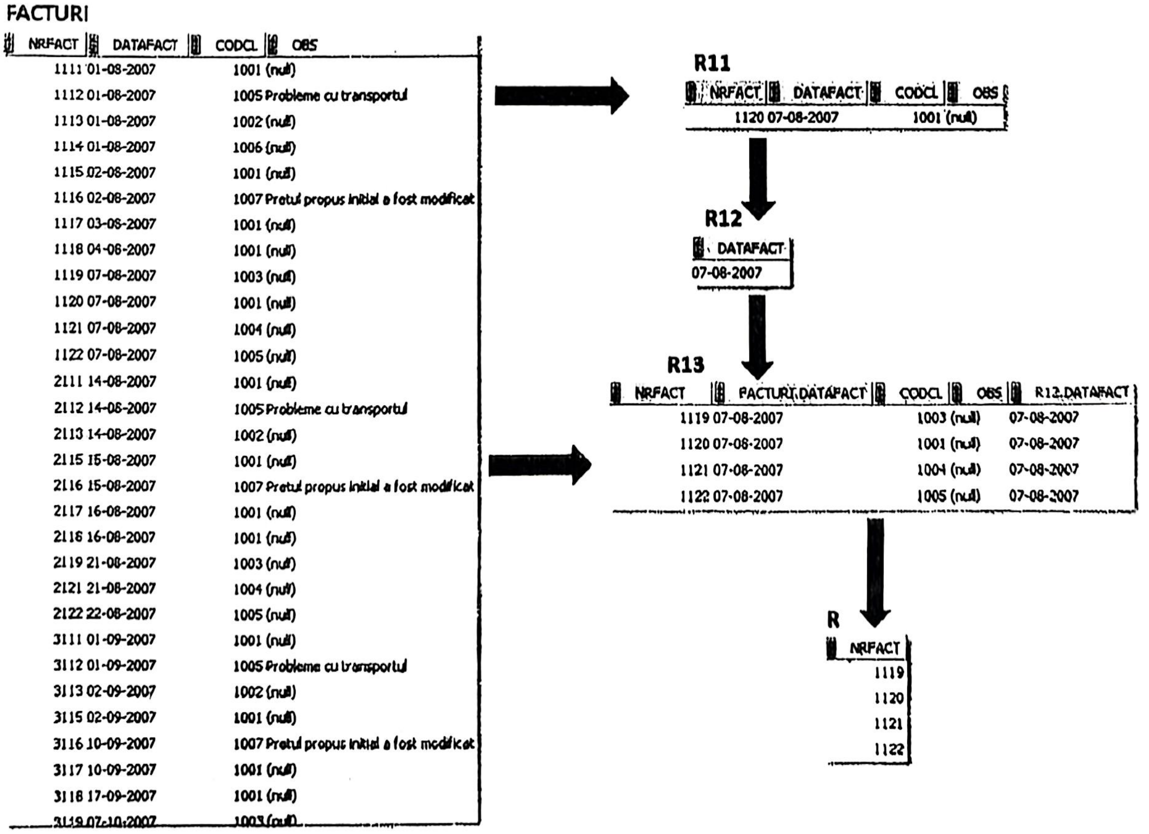
Exemplul 18. Ce clienți au cumpărat „Produs 2" și „Produs 3", dar nu au cumpărat „Produs 5"?

Bună întrebare! Lucrurile nu sunt însă atât de complicate precum par, deoarece folosim elemente din exemplul anterior (intersecţia) plus operatorul... diferenţă.

R11 ← SELECŢIE (PRODUSE; DenPr = 'Produs 2')
R12 ← JONCŢIUNE (R11, LINIIFACT; CodPr)
R13 ← JONCŢIUNE (R12, FACTURI; NrFact)
R14 ← JONCŢIUNE (R13, CLIENŢI; CodCl)
R15 ← PROIECŢIE (R14; DenCl)
R16 ← SELECŢIE (PRODUSE; DenPr = 'Produs 3')
R17 ← JONCŢIUNE (R16, LINIIFACT; CodPr)
R18 ← JONCŢIUNE (R17, FACTURI; NrFact)
R19 ← JONCŢIUNE (R18, CLIENŢI; CodCl)
R20 ← PROIECŢIE (R19; DenCl)
R21 ← SELECŢIE (PRODUSE; DenPr = 'Produs 5')
R22 ← JONCŢIUNE (R21, LINIIFACT; CodPr)
R23 ← JONCŢIUNE (R22, FACTURI; NrFact)
R24 ← JONCŢIUNE (R23, CLIENŢI; CodCl)
R25 ← PROIECŢIE (R24; DenCl)
R ← R15 ∩ R20 - R25

Exemplul 19. Ce facturi au fost emise în aceeaşi zi cu factura 1120?

Spre deosebire de interogările de până acum, condiţia de selecţie este una indirectă. În prealabil, trebuie determinată ziua în care a fost întocmită factura cu numărul 1120.



FACTURI

NRFACT	DATAFACT	CODCL	OBS
1111	01-08-2007	1001	(null)
1112	01-08-2007	1005	Probleme cu transportul
1113	01-08-2007	1002	(null)
1114	01-08-2007	1006	(null)
1115	02-08-2007	1001	(null)
1116	02-08-2007	1007	Pretul propus initial a fost modificat
1117	03-08-2007	1001	(null)
1118	04-08-2007	1001	(null)
1119	07-08-2007	1003	(null)
1120	07-08-2007	1001	(null)
1121	07-08-2007	1004	(null)
1122	07-08-2007	1005	(null)
2111	14-08-2007	1001	(null)
2112	14-08-2007	1005	Probleme cu transportul
2113	14-08-2007	1002	(null)
2115	15-08-2007	1001	(null)
2116	15-08-2007	1007	Pretul propus initial a fost modificat
2117	16-08-2007	1001	(null)
2118	16-08-2007	1001	(null)
2119	21-08-2007	1003	(null)
2121	21-08-2007	1004	(null)
2122	22-08-2007	1005	(null)
3111	01-09-2007	1001	(null)
3112	01-09-2007	1005	Probleme cu transportul
3113	02-09-2007	1002	(null)
3115	02-09-2007	1001	(null)
3116	10-09-2007	1007	Pretul propus initial a fost modificat
3117	10-09-2007	1001	(null)
3118	17-09-2007	1001	(null)
3119	07-10-2007	1003	(null)

R11

NRFACT	DATAFACT	CODCL	OBS
1120	07-08-2007	1001	(null)

R12

DATAFACT
07-08-2007

R13

NRFACT	FACTURI.DATAFACT	CODCL	OBS	R12.DATAFACT
1119	07-08-2007	1003	(null)	07-08-2007
1120	07-08-2007	1001	(null)	07-08-2007
1121	07-08-2007	1004	(null)	07-08-2007
1122	07-08-2007	1005	(null)	07-08-2007

R

NRFACT
1119
1120
1121
1122

Figura 4.24. Facturile emise în aceeaşi zi cu 1120 – varianta „economicoasă"

Apoi trebuie extrase, din relația FACTURI, liniile pentru care DataFact are valoarea zilei facturii-reper. Soluția „economicoasă", a cărei logică este expusă în figura 4.24, este:

R11 ← SELECȚIE (FACTURI; NrFact = 1120)
R12 ← PROIECȚIE (R11; DataFact)
R13 ← JONCȚIUNE (FACTURI, R12; DataFact)
R ← PROIECȚIE (R13; NrFact)

O variantă „risipitoare" (însă la fel de corectă, ca rezultat) presupune auto-joncționarea tabelei FACTURI. Astfel spus, se joncționează două instanțe ale tabelei FACTURI după atributul DataFact.

R21 ← JONCȚIUNE (FACTURI F1, FACTURI F2; F1.DataFact=F2.DataFact)
R22 ← SELECȚIE (R21; F1.NrFact = 1120)
R ← PROIECȚIE (R22; F2.NrFact)

Fireşte, trebuie făcute nişte supoziții despre modul în care vor fi denumite atributele în R21 şi R22, întrucât fiecare apare de două ori, corespunzător celor două instanțe. Din fericire, algebra relațională este teoretică, aşa că ne luăm o anumită libertate, fără riscul represaliilor. Derularea operațiunilor este sugerată în figura 4.25.

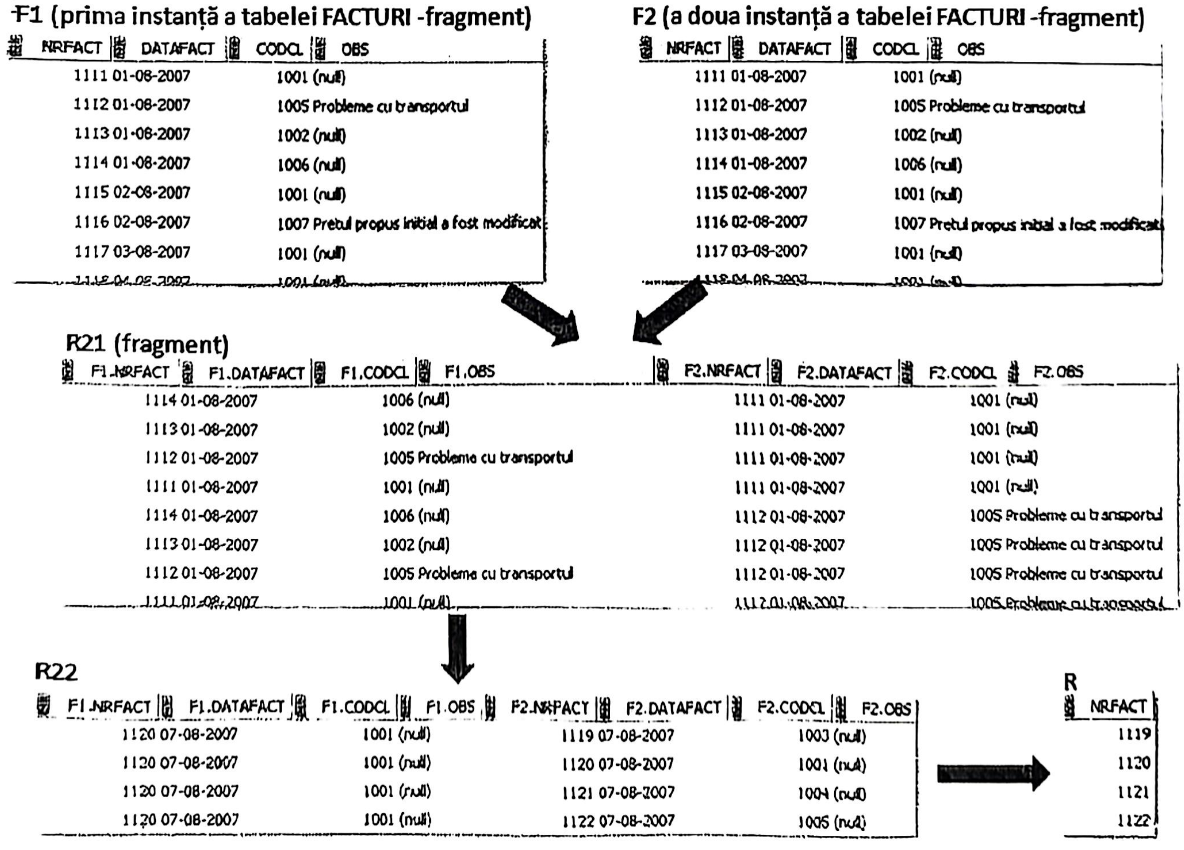
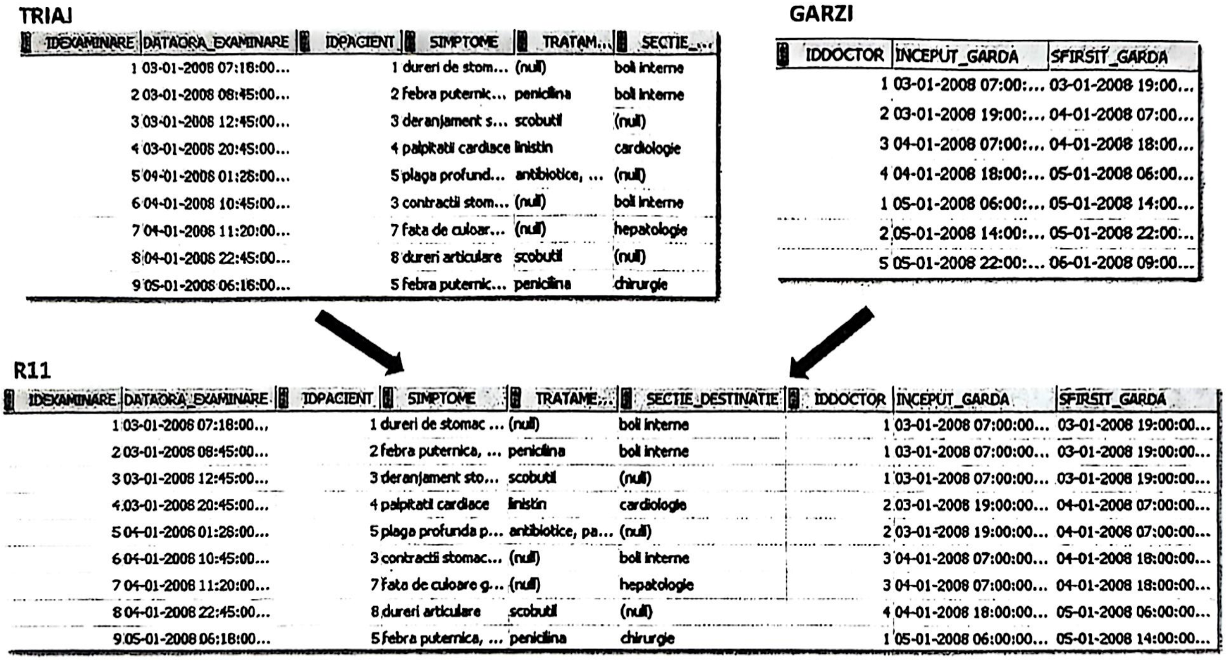


Figura 4.25. Facturile emise în aceeaşi zi cu 1120 – varianta „risipitoare"

Exemplul 20. Să se afişeze numele doctorului pentru fiecare examinare din triaj

Riscam să încheiem paragraful şi să nu dăm măcar un exemplu de theta--joncţiune. Nu întâmplător. Cazurile în care trebuie să recurgem la o theta--joncţiune (care nu este echijoncţiune) sunt ceva mai rare. Noi o să apelăm la cele câteva tabelele din baza de date a aplicaţiei *Camera de gardă* sau, mă rog, *Spitalul de urgenţă*. Ştiind că, pentru spitalul luat în discuţie, în orice moment, de gardă este un singur medic, soluţia presupune joncţionarea tabelei TRIAJ cu tabela GĂRZI. Interesant este că joncţionarea nu se bazează pe egalitatea valorilor dintre atributele DataExaminare şi vreunul dintre atributele Inceput_Garda, Sfarsit_Garda. DataExaminare reprezintă momentul efectiv al sosirii în triaj a bolnavului, în timp ce fiecare gardă este specificată printr-un interval dată/timp (vezi figura 4.26).

R11 ← JONCŢIUNE (TRIAJ, GARZI;
    TRIAJ.DataOra_Examinare >= GARZI.InceputGarda AND
    TRIAJ.DataOra_Examinare <= GARZI.SfarsitGarda)



TRIAJ

IDEXAMINARE	DATAORA_EXAMINARE	IDPACIENT	SIMPTOME	TRATAM...	SECTIE_...
1	03-01-2008 07:18:00...	1	dureri de stom...	(null)	boli interne
2	03-01-2008 08:45:00...	2	febra puternic...	penicilina	boli interne
3	03-01-2008 12:45:00...	3	deranjament s...	scobutil	(null)
4	03-01-2008 20:45:00...	4	palpitatii cardiace	linistin	cardiologie
5	04-01-2008 01:28:00...	5	plaga profund...	antibiotice, ...	(null)
6	04-01-2008 10:45:00...	3	contractii stom...	(null)	boli interne
7	04-01-2008 11:20:00...	7	fata de culoar...	(null)	hepatologie
8	04-01-2008 22:45:00...	8	dureri articulare	scobutil	(null)
9	05-01-2008 06:18:00...	5	febra puternic...	penicilina	chirurgie

GARZI

IDDOCTOR	INCEPUT_GARDA	SFIRSIT_GARDA
1	03-01-2008 07:00:...	03-01-2008 19:00...
2	03-01-2008 19:00:...	04-01-2008 07:00...
3	04-01-2008 07:00:...	04-01-2008 18:00...
4	04-01-2008 18:00:...	05-01-2008 06:00...
1	05-01-2008 06:00:...	05-01-2008 14:00...
2	05-01-2008 14:00:...	05-01-2008 22:00:...
5	05-01-2008 22:00:...	06-01-2008 09:00...

R11

IDEXAMINARE	DATAORA_EXAMINARE	IDPACIENT	SIMPTOME	TRATAME...	SECTIE_DESTINATIE	IDDOCTOR	INCEPUT_GARDA	SFIRSIT_GARDA
1	03-01-2008 07:18:00...	1	dureri de stomac ...	(null)	boli interne	1	03-01-2008 07:00:00...	03-01-2008 19:00:00...
2	03-01-2008 08:45:00...	2	febra puternica, ...	penicilina	boli interne	1	03-01-2008 07:00:00...	03-01-2008 19:00:00...
3	03-01-2008 12:45:00...	3	deranjament sto...	scobutil	(null)	1	03-01-2008 07:00:00...	03-01-2008 19:00:00...
4	03-01-2008 20:45:00...	4	palpitatii cardiace	linistin	cardiologie	2	03-01-2008 19:00:00...	04-01-2008 07:00:00...
5	04-01-2008 01:28:00...	5	plaga profunda p...	antibiotice, pa...	(null)	2	03-01-2008 19:00:00...	04-01-2008 07:00:00...
6	04-01-2008 10:45:00...	3	contractii stomac...	(null)	boli interne	3	04-01-2008 07:00:00...	04-01-2008 18:00:00...
7	04-01-2008 11:20:00...	7	fata de culoare g...	(null)	hepatologie	3	04-01-2008 07:00:00...	04-01-2008 18:00:00...
8	04-01-2008 22:45:00...	8	dureri articulare	scobutil	(null)	4	04-01-2008 18:00:00...	05-01-2008 06:00:00...
9	05-01-2008 06:18:00...	5	febra puternica, ...	penicilina	chirurgie	1	05-01-2008 06:00:00...	05-01-2008 14:00:00...

Figura 4.26.Theta-joncţiune dintre TRIAJ şi GĂRZI

Soluţia continuă astfel:

R12 ← JONCŢIUNE (R11, DOCTORI; IdDoctor)
R13 ← JONCŢIUNE (R12, PACIENŢI; IdPacient)
R ← PROIECŢIE (R13, DataOra_Examinare, NumePacient, NumeDoctor)

4.4.5. Joncţiunea externă

Ideea de bază a joncţiunii externe este de a include în rezultat şi tupluri din una din relaţii sau din ambele relaţii, care prezintă valori ale atributului de legătură ce *nu* se regăsesc în cealaltă relaţie. Dacă precedentele tipuri de joncţiune sunt comutative, în cazul joncţiunii externe trebuie specificat din care relaţie se extrag liniile fără corepondent în cealaltă relaţie. De aceea, există *joncţiune externă la stânga*

şi *joncţiune externă la dreapta*. La acestea se adaugă *joncţiunea externă totală* (denumită şi *plină* sau *deplină*) care reprezintă reuniunea celor două. Apelând la aceleaşi tabele „abstracte", R1 şi R2, diferenţa dintre joncţiunea internă (echi-joncţiunea) şi cele trei tipuri de joncţiune externă apare cu mai multă claritate în figura 4.27.

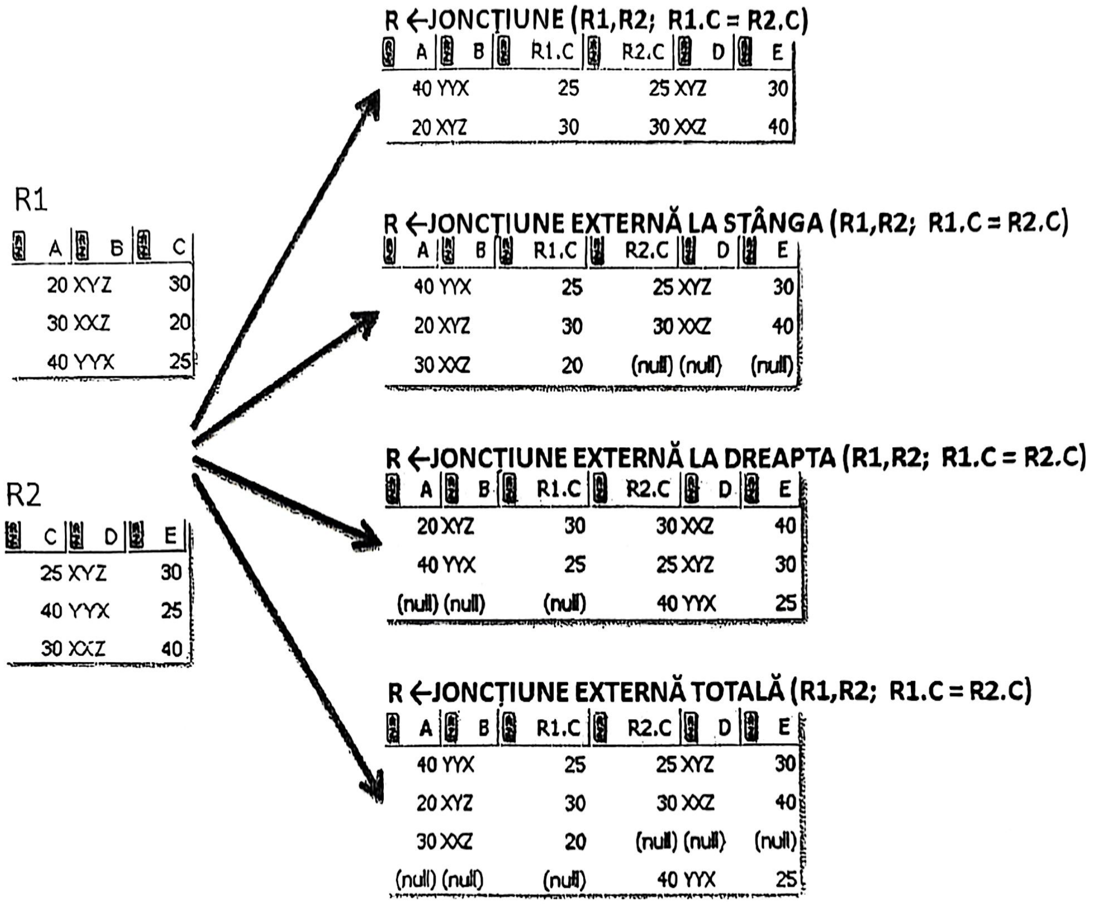


Figura 4.27. Diferenţa dintre echijoncţiune şi joncţiunile externe

Liniile dintr-o tabelă care prezintă valori ale atributului de legătură fără corespondent în cealaltă tabelă sunt completate în rezultat cu valori NULL.

Exemplul 21. Care sunt codurile poştale care nu fac parte din adresa niciunui client?

Soluţia 1 se bazează pe diferenţa dintre tabela tuturor codurilor poştale preluate din CODURI_POŞTALE şi tabela codurilor la care există cel puţin un client (din CLIENŢI).

R31 ← PROIECŢIE (CODURI_POŞTALE; CodPost)
R32 ← PROIECŢIE (CLIENŢI; CodPost)
R33 ← R31 - R32
R ← JONCŢIUNE (R33, CODURI_POŞTALE; CodPost)

Soluţia 2 utilizează proaspăta joncţiune externă.

R31 ← JONCŢIUNE EXTERNĂ LA STÂNGA (CODURI_POŞTALE, CLIENŢI;

CODURI_POŞTALE.CodPost = CLIENŢI.CodPost)
R ← SELECŢIE (R31; CodCl IS NULL)

Pentru a identifica localităţile fără clienţi, s-au extras, din joncţiunea externă la stânga a relaţiilor CODURI_POŞTALE şi CLIENŢI, numai liniile în care unul din atributele preluate din CLIENŢI (preferabil cheia primară) are valoarea NULL.

CODURI_POSTALE

CODPOST	LOC	JUD
700505	Iasi	IS
700510	Iasi	IS
700515	Iasi	IS
701150	Pascani	IS
706500	Vaslui	VS
706510	Vaslui	VS
705300	Focsani	VN
705310	Focsani	VN
706400	Birlad	VS
705800	Suceava	SV
705550	Roman	NT
701900	Timisoara	TM

CLIENŢI

CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	(null)
1002	Client 2 SA	R1002	(null)	700505	032-212121
1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	035-222222
1004	Client 4	(null)	Sapientei, 56	701150	(null)
1005	Client 5 SRL	R1005	(null)	701900	056-111111
1006	Client 6 SA	R1006	Pacientei, 33	705550	(null)
1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	056-121212

R31

CODURI_POSTALE.CODPOST	LOC	JUD	CODCL	DENCL	CODFISCAL	ADRESA	CLIENTI.CODPOST	TELEFON
700505	Iasi	IS	1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	(null)
700505	Iasi	IS	1002	Client 2 SA	R1002	(null)	700505	032-212121
706500	Vaslui	VS	1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	035-222222
701150	Pascani	IS	1004	Client 4	(null)	Sapientei, 56	701150	(null)
701900	Timisoara	TM	1005	Client 5 SRL	R1005	(null)	701900	056-111111
705550	Roman	NT	1006	Client 6 SA	R1006	Pacientei, 33	705550	(null)
701900	Timisoara	TM	1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	056-121212
705300	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
706400	Birlad	VS	(null)	(null)	(null)	(null)	(null)	(null)
705310	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705800	Suceava	SV	(null)	(null)	(null)	(null)	(null)	(null)
700515	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
700510	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
706510	Vaslui	VS	(null)	(null)	(null)	(null)	(null)	(null)

R

CODURI_POSTALE.CODPOST	LOC	JUD	CODCL	DENCL	CODFISCAL	ADRESA	CLIENTI.CODPOST	TELEFON
705300	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
706400	Birlad	VS	(null)	(null)	(null)	(null)	(null)	(null)
705310	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705800	Suceava	SV	(null)	(null)	(null)	(null)	(null)	(null)
700515	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
700510	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
706510	Vaslui	VS	(null)	(null)	(null)	(null)	(null)	(null)

Figura 4.28. Folosirea joncţiunii externe – exemplul 21

4.4.6. Semijoncţiunea

Semijoncţiunea este unul din cele mai „marginalizate" tipuri de joncţiune. Implementarea sa în SGBD-urile comerciale este extrem de rară. A fost introdusă din dorinţa de a optimiza procesul de interogare (consultare). Calculul semijoncţiunii a două tabele presupune *selectarea numai a liniilor din prima tabelă care apar în joncţiune cu linii din a doua tabelă* – vezi figura 4.29.

Fireşte, operatorul nu este comutativ. Implementarea sa în SQL este una destul de facilă, după cum vom vedea în capitolele următoare.

Exemplul 22. Care sunt codurile poştale (plus denumirea localităţii şi indicativul judeţului) în care există măcar un client?

Soluția 1 - bazată pe echijoncțiune:
R1 ← PROIECȚIE (CLIENȚI; CodPost)
R2 ← JONCȚIUNE (R1, CODURI_POȘTALE; CodPost)
R ← PROIECȚIE (R2; CodPost, Loc, Jud)

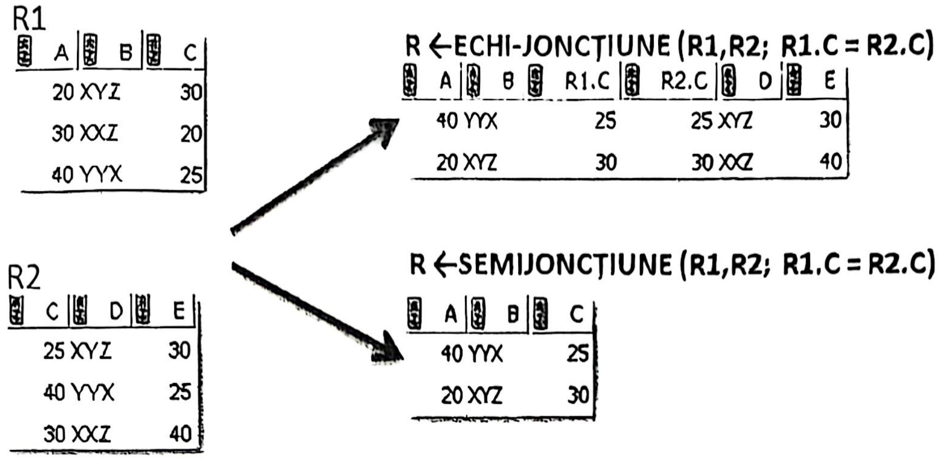


Figura 4.29. Diferența dintre echijoncțiune și semijoncțiune

Soluția 2 - bazată pe semijoncțiune:
R ← SEMIJONCȚIUNE (CODURI_POȘTALE, CLIENȚI; CodPost)

4.4.7. Diviziunea

Este cel mai complex și mai greu de explicat dintre operatorii prezentați în acest capitol. Codd l-a imaginat ca operator invers al produsului cartezian. Pentru a-l defini, se pornește de la două relații RD1(X,Y) și RD2(Y); prima are, care va să zică, două atribute sau grupe de atribute, notate X și Y, în timp ce a doua numai atributul sau grupul de atribute notat cu Y (definit pe același domeniu ca și în relația RD1). O primă restricție: relația RD2(Y), fiind numitorul diviziunii, nu trebuie să fie vidă.

Diviziunea relațională RD1 ÷ RD2 are ca rezultat o relație definită ca ansamblul subtuplurilor RD1(X) pentru care produsul (lor) cartezian cu RD2(Y) este un subansamblu al RD1(X,Y). Rezultatul expresiei RD1 ÷ RD2 reprezintă câtul diviziunii, fiind o relație ce poate fi notată RD3(X). Într-o altă formulare, xi ∈ R3 dacă și numai dacă ∀yi ∈ Y ∈ R2 → ∃ (xi, yi) ∈ R1. Pentru simplificarea prezentării, în continuare am considerat X și Y două atribute, deși, după cum reiese din preambul, acestea pot fi grupe (ansambluri) de atribute. Să examinăm elementele din figura 4.30.

„Calcularea" relației R ← RD1 ÷ RD2 este sinonimă cu rezolvarea problemei: care dintre x1, x2, x3, x4 și x5 apar în RD1, în tupluri împreună cu toate valorile lui Y din RD2, respectiv y1, y2, y3, y4 și y5?

Se parcurg pe rând valorile xi ale atributului X din relația RD1:

- x1 apare cu y1 (în tuplul 1), cu y2 (în tuplul 4), cu y3 (în tuplul 7), cu y4 (în tuplul 10) şi cu y5 (în tuplul 13). Deci x1 îndeplineşte condiţia şi va fi inclus în relaţia R;
- x2 apare cu y1 (în tuplul 2), dar nu apare cu y2 - nu va face parte din R;
- x3 apare cu y1 (în tuplul 3), cu y2 (în tuplul 5), cu y3 (în tuplul 8), cu y4 (în tuplul 11) şi cu y5 (în tuplul 15) - îndeplineşte condiţia şi va fi tuplu în R;
- x4 nu apare cu y1 - nu va face parte din R;
- x5 nu apare cu y1 - nu va face parte din R.

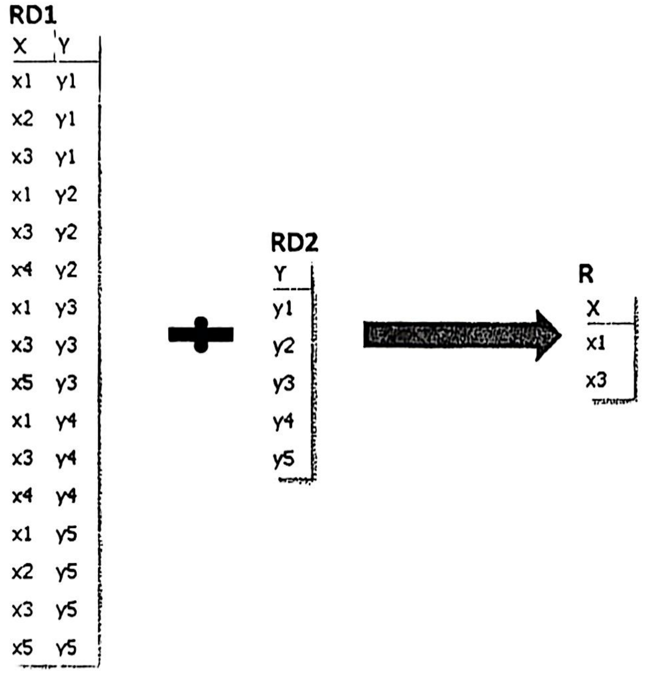


Figura 4.30. Diviziunea relaţională

În urma raţionamentului de mai sus, tabela R va fi alcătuită din două tupluri. Deşi pare un operator ceva mai metafizic, diviziunea relaţională este deosebit de utilă pentru formularea consultărilor în care apare clauza „∀" („oricare ar fi" sau „pentru toate").

Exemplul 23. Care sunt clienţii pentru care există cel puţin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007?

Într-o altă formulare, ne interesează clienţii care au cumpărat „câte ceva" în toate zilele în care s-au efectuat vânzări (s-a întocmit măcar o factură) din perioada 10-30 septembrie 2007? Prin urmare, câtul va fi o tabelă cu singur atribut, DenCl (denumirea clientului), iar divizorul va fi o relaţie alcătuită numai din atributul DataFact (conţine toate zilele în care s-au efectuat vânzări). Dacă am merge pe calapodul prezentat, am putea nota R(DenCl), RD2(DataFact). Cunoscând structura câtului şi a divizorului, putem determina structura tabelei-deîmpărţit: RD1(DenCl, DataFact). Relaţia RD1 va conţine denumirile clienţilor şi zilele în care există măcar o factură pentru clientul respectiv. Soluţia poate fi redactată în următorii paşi:

- construirea relaţiei deîmpărţit:

  R11 ← SELECŢIE (FACTURI; DataFact >= '2007-09-10' AND

DataFact <= '2007-09-30')
R12 ← JONCȚIUNE (CLIENTI, R11; CodCl)
RD1 ← PROIECȚIE (R12; DenCl, DataFact)

- construirea relației numitor:
  RD2 ← PROIECȚIE (R11; DataFact)
- în fine, apoteoza:
  R ← RD1 ÷ RD2

Relațiile implicate în această soluție (cu excepțiile FACTURI și CLIENȚI) sunt prezentate în figura 4.31.

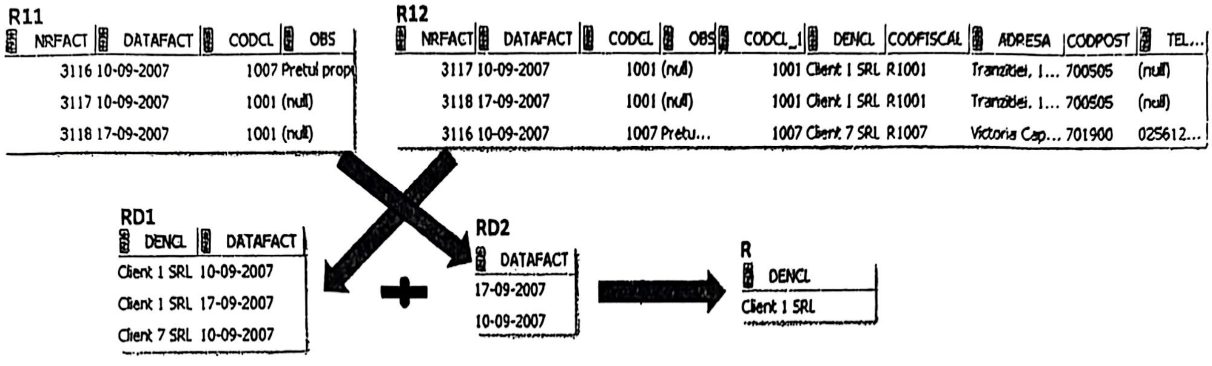


Figura 4.31. Diviziunea relațională – exemplul 23

Poate o să spuneți că n-a meritat efortul, dar acest gen de soluții se aplică la o largă gamă de probleme. Să discutăm câteva spețe.

Exemplul 24. În ce zile s-au vândut și produsul cu denumirea „Produs 1" și cel cu denumirea „Produs 2"?

Vă gândiți, probabil, că ați mai văzut undeva acest enunț. Este chiar textul problemei de la exemplul 17. La acel moment, am formulat două soluții: una „clasică", bazată pe intersecție și una mai neconvențională ce utilizează joncțiunea. Adăugăm la acestea o alta, încadrabilă probabil tot în categoria „neconvenționale".

Soluția 3

R11 ← SELECȚIE (PRODUSE; DenPr = 'Produs 1' OR DenPr = 'Produs 2')
R12 ← JONCȚIUNE (R11, LINIIFACT; CodPr)
R13 ← JONCȚIUNE (R12, FACTURI; NrFact)
R1 ← PROIECȚIE (R13; DataFact, CodPr)
R2 ← PROIECȚIE (R11; CodPr)
R3 ← R1 ÷ R2

Relația divizor este alcătuită din două tupluri ce conțin codurile celor două produse. Deîmpărțitul este alcătuit din atributele DataFact și CodProd și conține toate produsele vândute în fiecare din zilele de facturare.

Exemplul 25. Ce facturi au fost emise în aceeași zi cu factura 1120?

Deşi reprezintă un regres vizibil faţă de nivelul interogărilor precedente, revenim discret la enunţul exemplului 19. Pentru această problemă se poate încropi şi o variantă de rezolvare bazată pe diviziunea relaţională.
R1 ← PROIECŢIE (FACTURI; NrFact, DataFact)
R2' ← SELECŢIE (FACTURI; NrFact = 1120)
R2 ← PROIECŢIE (R2'; DataFact)
R ← DIVIZIUNE (R1, R2)

Exemplul 26. Ce produse au fost vândute tuturor clienţilor?
R11 ← JONCŢIUNE (LINIIFACT, FACTURI; NrFact)
R1 ← PROIECŢIE (R11; CodPr, CodCl)
R2 ← PROIECŢIE (CLIENŢI; CodCl)
R3 ← DIVIZIUNE (R1, R2)
R ← JONCŢIUNE (R3, PRODUSE; CodPr)

Exemplul 27. Care sunt facturile care conţin măcar produsele din factura 1117?
Şi acesta este un gen de probleme pentru care diviziunea relaţională este o binecuvântare. În tabela LINIIFACT putem observa că factura 1117 consemnează vânzarea a două produse ce au codurile 1 şi 2. Aşadar, trebuie să depistăm toate facturile în care apar măcar produsele 1 şi 2, eventual alte produse. Relaţia „numărător" R1 conţine două atribute, NrFact (X-ul) şi CodPr (Y-ul), iar tuplurile sale sunt, de fapt, cele ale tabelei LINIIFACT. Relaţia „numitor" R2 conţine codurile produselor (CodPr) prezentate în factura-etalon 1117.
R1 ← PROIECŢIE (LINIIFACT; NrFact, CodPr)
R21 ← SELECŢIE (LINIIFACT; NrFact = 1117)
R2 ← PROIECŢIE (R21; CodPr)
R ← DIVIZIUNE (R1, R2)

Schema de funcţionare a soluţiei este ilustrată în figura 4.32.
Diviziunea relaţională nu este un operator fundamental; funcţionalitatea sa poate fi realizată prin combinarea operatorilor: produs cartezian, diferenţă şi proiecţie. Se reiau în discuţie atât tabelele RD1 şi RD2 din figura 4.30, cât şi problema: *care dintre valorile x_i apar în tupluri, în RD1, cu* toate *valorile y_j din RD2?*

O soluţie a problemei poate fi constituită din următorii paşi:
R11 ← PROIECŢIE (RD1; X)
R12 ← R11 ⊗ RD2. Tabela R12 cuprinde toate tuplurile posibile (x_i, y_j).

R13 ← R12 - RD1. Interesează ce tupluri din R12 lipsesc în R1. Valorile x_i din R13 *nu* apar în RD1 în combinaţie cu *toate* valorile y_j din RD2. Se elimină dublurile prin:

R14 ← PROIECŢIE (R13; X). Valorile x_i din R14 sunt cele care *nu* prezintă tupluri obţinute prin combinaţii cu toate valorile y_j. Deoarece interesează cele care *prezintă* combinaţiile respective, rezultatul se obţine prin diferenţa:
R ← R11 - R14.

R1 (parţial)

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1111	1	1	50	1000
1111	2	2	75	1050
1111	3	5	500	7060
1112	1	2	80	1030
1112	2	3	40	750
1113	1	2	100	975
1114	1	2	70	1070
1114	2	4	30	1705
1114	3	5	700	7064
1115	1	2	150	925
1116	1	2	125	930
1117	2	1	100	950
1117	1	2	100	1000
1118	2	1	150	930
1118	1	2	30	1100
1119	1	2	35	1090
1119	2	3	40	700
1119	3	4	50	1410
1119	4	5	750	6300
1120	1	2	80	1120
1121	2	2	100	1050

R2

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1117	1	2	100	1000
1117	2	1	100	950

R

NRFACT
1111
1117
1118
2111
2117
2118
3111
3117
3118

Figura 4.32. Diviziunea relaţională – exemplul 27

Figura 4.33 aduce, cât de cât, un plus de claritate.

RD1

X	Y
x1	y1
x2	y1
x3	y1
x1	y2
x3	y2
x4	y2
x1	y3
x3	y3
x5	y3
x1	y4
x3	y4
x4	y4
x1	y5
x2	y5
x3	y5
x5	y5

R11 ←PROIECŢIE (RD1; X)

X
x1
x3
x2
x5
x4

RD2

Y
y1
y2
y3
y4
y5

R12 ← R11⊗ RD2

X	Y
x1	y1
x1	y2
x1	y3
x1	y4
x1	y5
x3	y1
x3	y2
x3	y3
x3	y4
x3	y5
x2	y1
x2	y2
x2	y3
x2	y4
x2	y5
x5	y1
x5	y2
x5	y3
x5	y4
x5	y5
x4	y1
x4	y2
x4	y3
x4	y4
x4	y5

R13 ←R12 – RD1

X	Y
x2	y2
x2	y3
x2	y4
x4	y1
x4	y3
x4	y5
x5	y1
x5	y2
x5	y4

R14 ←PROIECŢIE (R13; X)

X
x2
x4
x5

R ← R11 – R14)

X
x1
x3

Figura 4.33. Simularea diviziunii prin alţi operatori relaţionali

Dintre operatorii prezentaţi pe parcursul ultimelor două paragrafe, selecţia, proiecţia, produsul cartezian, reuniunea şi diferenţa sunt operatori *fundamentali* sau *ireductibili*, în sensul că niciunul nu poate fi definit prin intermediul celorlalţi.

Joncţiunea, intersecţia şi diviziunea sunt operatori *derivaţi*. Joncţiunea se exprimă cu ajutorul produsului cartezian urmat de selecţie; intersecţia se exprimă cu ajutorul diferenţei, conform relaţiei **R1 ∩ R2 = R1 – (R1 – R2)**; diviziunea poate fi exprimată, aşa cum am văzut ceva mai sus, prin intermediul produsului cartezian, proiecţiei şi diferenţei.

4.5. Notaţii şi reprezentări grafice ale interogărilor

Încă de la începutul prezentării operatorilor algebrei relaţionale, precizam că notaţia pe care o vom folosi este una din cele mai comode (nu şi cea mai uzuală). Modelul relaţional este, în ciuda materialului de faţă, unul matematizat. Algebra relaţională este, aşa cum îi spune şi numele, riguros formalizată. De aceea, şi pentru un look mai ştiinţific, majoritatea lucrărilor folosesc o notaţie matematică sau... aproximativ matematică.

4.5.1. Notaţia matematică

Există o multitudine de lucrări care utilizează această notaţie. În cazul de faţă, sursa principală o constituie traducerea franceză o unei lucrări mai vechi de Korth şi Silberschatz. Notaţia prezentată foloseşte, pe lângă clasicele: ∪ - reuniune, ∩ - intersecţie, × - produs cartezian, ÷ - diviziune şi simboluri din alfabetul grecesc, după cum urmează: pentru selecţie: σ_P (R), unde P este un predicat aplicat asupra relaţiei R, pentru proiecţie: Π_S (R), unde S este o listă de atribute din R, pentru joncţiune: $R1 \bowtie_{R1.A>R2.C} R2$ – indică (theta)joncţiunea relaţiilor R1 şi R2 prin predicatul: R1.A > R2.C.

Exemple

Soluţia de la exemplul 7:

R1 ← SELECŢIE (CLIENTI; DenCl = "Client 2 SA")

R2 ← PROIECŢIE (R1; Telefon)

poate fi transcrisă în notaţia matematică după cum urmează:

$$\prod_{Telefon}(\sigma_{DenCl="Client2\ SA"}(CLIENTI))$$

După cum se observă, întreaga interogare se scrie într-o singură expresie.

Soluţia de la exemplul 9:

R1 ← SELECŢIE (LINIIFACT; NrFact = 1111)

R2 ← PROIECŢIE (R1; CodPr)

R3 ← SELECŢIE (LINIIFACT; NrFact = 1117)

R4 ← PROIECŢIE (R3; CodPr)

R5 ← R2 ∩ R4:

$$\prod_{CodPr}(\sigma_{NrFact=1111}(LINIIFACT)) \bigcap \prod_{CodPr}(\sigma_{NrFact=1117}(LINIIFACT))$$

Soluția 1 de la exemplul 14:

R1 ← JONCȚIUNE (LOCALITATI, JUDETE; Jud)

R ← SELECȚIE (R1; Regiune = "Banat")

$$\sigma_{Regiune="Banat"}(LOCALITATI \rhd\lhd_{LOCALITATI.Jud=JUDETE.Jud} JUDETE)$$

Este drept, atunci când este de reprezentat o joncțiune naturală, se poate recurge la o redactare simplificată:

$$\sigma_{Regiune="Banat"}(LOCALITATI \rhd\lhd JUDETE)$$

O altă observație, ține de faptul că optimizarea interogărilor nu prezintă aproape deloc importanță.

Soluția de la exemplul 16

R1 ← SELECȚIE (PRODUSE; DenPr = "Produs 1")

R2 ← JONCȚIUNE (R1, LINIIFACT; CodPr)

R3 ← PROIECȚIE (R2; NrFact)

R4 ← SELECȚIE(FACTURI; DataFact >= 03/08/2000 AND DataFact <= 03/08/2000)

R5 ← JONCȚIUNE (R3, R4; NrFact)

R6 ← JONCȚIUNE (R5, CLIENTI; CodCl)

R7 ← PROIECȚIE (R6; CodPost)

R8 ← JONCȚIUNE (R7, LOCALITATI; CodPost)

R9 ← PROIECȚIE (R8; Jud)

R10 ← JONCȚIUNE (R9, JUDETE; Jud)

R ← PROIECȚIE (R10; Judet):

$$\prod_{Judet}(\sigma_{DenPr="Produs 1" \wedge DataFact>=03/08/2000 \wedge DataFact<=05/08/2000}(PRODUSE \rhd\lhd$$
$$LINIIFACT \rhd\lhd FACTURI \rhd\lhd CLIENTI \rhd\lhd LOCALITATI \rhd\lhd JUDETE))$$

Nu spun că acest stil de notare ar fi din cale afară de dificil, dar eu, unul, consider că varianta „noastră" de redactare este mult mai lejeră.

4.5.2. Gramatica BNF

Notația următoare este preluată din ediția a 4-a a lucrării lui C.J. Date și, cu oarecare amendamente din partea lui Date, reprezintă gramatica *Backus Naur Form*. Avantajele acestei notații țin, în primul rând, de utilizarea unor cuvinte din limba engleză, în locul abstractelor simboluri matematice. De asemenea, interogările se scriu liniar, fiind astfel mai lizibile. Pentru a indica precedența operațiilor se folosesc parantezele. Un avantaj major față de notația folosită pe parcursul acestui capitol ține de faptul că pune în evidență modul în care operațiile sunt incluse

unele în altele, fiind mai aproape şi de logica SQL (care foloseşte o singura frază SELECT în care, eventual, sunt incluse subconsultări).

Fără a intra în detalii privind gramatica BNF tratată în extenso în lucrarea autorului american, amintim că reprezentarea operatorilor se face astfel:

- *reuniune*: R1 UNION R2;
- *intersecţie*: R1 INTERSECT R2;
- *diferenţă*: R1 MINUS R2;
- *produs cartezian*: R1 TIMES R2;
- *selecţie*, denumită şi *restricţie*: R WHERE *P*, unde *P* este un predicat aplicat asupra relaţiei R;
- *proiecţie*: R [A, B, ... Z], unde A, B, ..., Z sunt atribute ale R;
- *theta-joncţiunea* se reprezintă diferit de *joncţiunea naturală*, ca un operator compus dintr-un produs cartezian urmat de o selecţie. Astfel, pentru (theta-) joncţiunea relaţiilor R1 şi R2 prin predicatul: R1.A > R2.C, se foloseşte notaţia: (R1 TIMES R2) WHERE R1.A > R2.C;
- pentru *joncţiunea naturală* lucrurile sunt mai simple; astfel, joncţiunea naturală dintre R1 şi R2 prin atributul C (atributului comun) se simbolizează, pur şi simplu: R1 JOIN R2;
- *diviziune*: R1 DIVIDED BY R2.

Exemple

Soluţia de la exemplul 7:

(CLIENTI WHERE DenCl = "Client 2 SA") [Telefon]

Soluţia de la exemplul 9:

((LINIIFACT WHERE NrFact = 1111) [CodPr]) INTERSECT

((LINIIFACT WHERE NrFact = 1111) [CodPr])

Soluţia 1 de la exemplul 14:

((JUDETE JOIN CODURI_POSTALE) WHERE Regiune = "Banat")

Soluţia de la exemplul 16

(((((((PRODUSE WHERE DenPr = "Produs 1")JOIN LINIIFACT) [NrFact]) JOIN FACTURI) WHERE DataFact >= 03/08/2000 AND DataFact <= 03/08/2000) JOIN CLIENTI) JOIN CODURI_POSTALE) JOIN JUDETE) [Judet])

Probabil cel mai mare dezavantaj al ecestei notaţii ţine de faptul că s-ar putea să greşim la număratul parantezelor...

4.5.3. Notaţii diverse

Literatura de specialitate a propus diverse notaţii, mai mult sau mai puţin apropiate de cele prezentate până acum. Dacă e să ne referim, numai la cărţile publicate la noi (bazate, ca şi prezenta, pe lucrări de circulaţie internaţională), putem aminti:

- pentru reuniune:
  R1 $\cup$ R2

OR (R1, R2)
APPEND (R1, R2)
UNION (R1, R2)
REUNIUNE (R1,R2)

- pentru intersecție:
  R1 ∩ R2
  AND (R1, R2)
  INTERSECT (R1, R2)
  INTERSECȚIE (R1,R2)

- pentru diferență:
  R1 - R2
  REMOVE (R1, R2)
  MINUS (R1, R2)
  DIFERENȚĂ (R1,R2)

- pentru produs cartezian:
  R1 x R2
  PRODUCT (R1, R2)
  TIMES (R1, R2)
  PROD (R1,R2)

- pentru selecție:
  σcondiție (R)
  R [condiție]
  RESTRICT (R, condiție)
  SEL (R, condiție)

- pentru proiecție:
  ΠA, B, C,...Z (R)
  R [A,B, ..., Z]
  PROJECT (R, A, B, ...)
  PROI (R, A,B, ..., Z)

- pentru joncțiune:
  $R1 \underset{\text{conditie}}{\bowtie} R2$
  JOIN (R1, R2, condiție)

- pentru diviziune:
  R1 ÷ R2
  DIVISION (R1, R2)

4.5.4. Reprezentarea grafică a interogărilor

În multe lucrări de specialitate sunt utilizate o serie de simboluri grafice asociate operatorilor relaţionali. Dintre acestea, cele mai importante sunt prezentate în figura 4.34. Aceste simboluri permit reprezentarea, în condiţii de lizibilitate sensibil îmbunătăţită, a logicii de derulare a unei interogări formulate prin operatorii relaţionali.

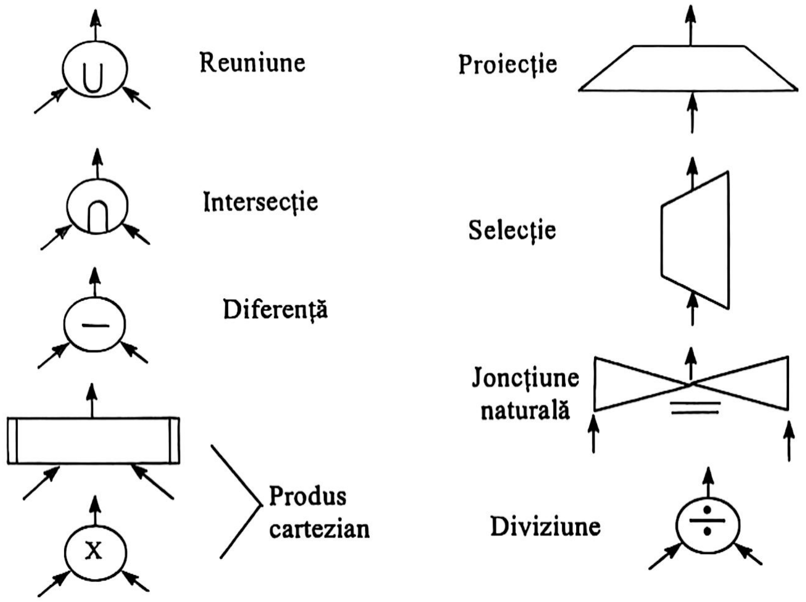


Figura 4.34. Simboluri pentru reprezentarea grafică a operatorilor algebrei relaţionale

Pentru ilustrarea dispunerii simbolurilor în concordnanţă cu logica soluţiei, reluăm cele patru exemple folosite în acest paragraf.

Exemple

Soluţia de la *exemplul 7* - figura 4.35:

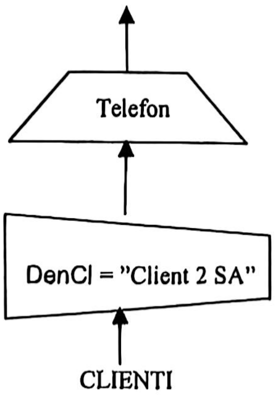


Figura 4.35. Reprezentare grafică – soluţia pentru exemplul 7

Soluția de la exemplul 9 – figura 4.36:

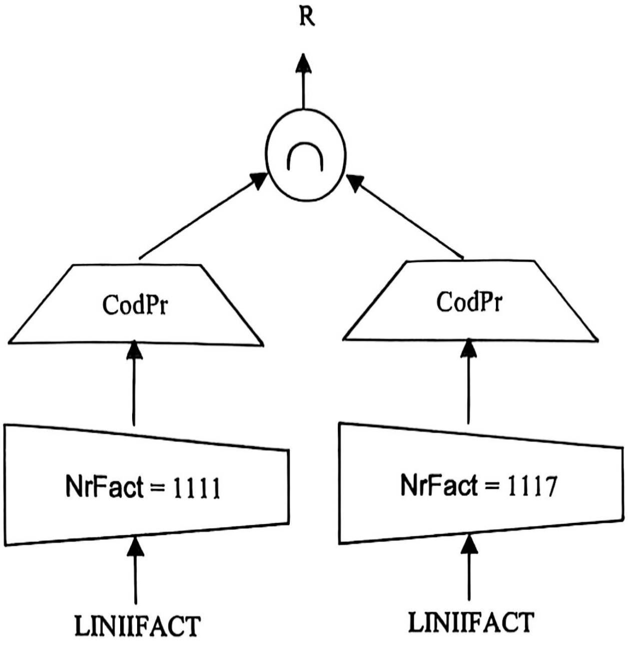


Figura 4.36. Reprezentare grafică – soluția pentru exemplul 9

Soluția 1 de la exemplul 14 – figura 4.37.

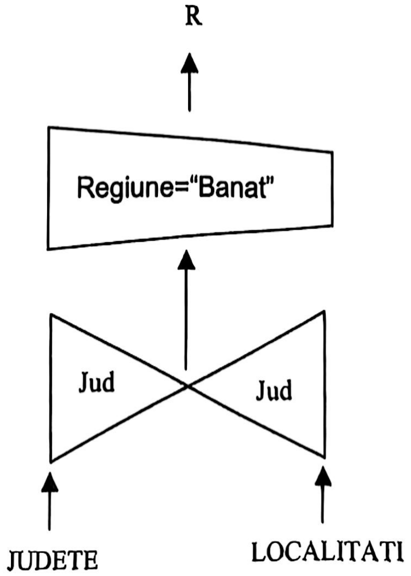


Figura 4.37. Reprezentare grafică – soluția 1 pentru exemplul 14

Capitolul 5
Sintaxa de bază a comenzii SELECT

Fără îndoială, cea mai gustată parte din SQL este cea legată de interogarea bazelor de date, adică de obținerea de informații dintre cele mai diverse, prin prelucrări, grupări etc. În SQL, o interogare se formulează printr-o comandă[1] SELECT care are un format pe cât de simplu, pe atât de flexibil și puternic. Deși bazat pe algebra relațională, SQL-ul este cu mult mai complex. Este motivul pentru care paralela cu operatorii algebrici relaționali poate fi făcută doar până la un punct.

Din perspectiva prezentei lucrări, obiectivul principal al SQL constă în a oferi utilizatorului mijloacele necesare formulării unei consultări numai prin descrierea rezultatului dorit, cu ajutorul unei aserțiuni (expresie logică), fără a fi necesară și explicitarea modului efectiv în care se face căutarea în BD. Altfel spus, *utilizatorul califică (specifică) rezultatul*, iar *sistemul se ocupă de procedura de căutare.*

Prin comparație cu algebra relațională, SQL este nu numai mai generos, dar și-a luat o serie de libertăți care au stârnit mânia multor „puriști" ai modelului relațional, cum ar fi E.F. Codd (vezi [Codd 1989]), C.J. Date (vezi [Date 2004/2005]) și F. Pascal[2]. Vom sublinia pe parcurs unele „devieri" SQL-istice. Unul dintre elementele-cheie care sugerează sorgintea relațională a SQL-ului este orientarea sa pe tupluri. Comanda SELECT pe care o vom întoarce pe toate părțile începând cu acest capitol, va obține, dintr-una, două sau mai multe tabele, un rezultat tabelar (o tabelă care e, de obicei, și anonimă, și temporară). SQL-ul nu lucrează la nivel de linie (individuală), ci de set de tupluri.

5.1. Trei clauze și un rezultat

Sintaxa frazei (comenzii) SELECT din SQL este grozav de simplă. Cele trei clauze principale sunt SELECT, FROM și WHERE[3], dintre care numai primele două sunt obligatorii[4]. Presupunând că am dispune de o bază de date în care tabela STUDENȚI conține toți studenții facultății, pentru a obține lista celor înscriși în anul 3 la specializarea Informatică Economică, comanda SELECT are sintaxa următoare:

SELECT NumePrenume

1. Vom vedea că sunt atât de multe opțiuni ce pot fi incluse într-o comandă SELECT, încât aceasta se poate întinde pe multe linii. Este, probabil, unul dintre motivele pentru care se folosește termenul *frază* SELECT.
2. Vezi [Pascal 2000], dar și site-ul lui Fabian unde vitriolul este la el acasă: *www.dbdebunk.com*.
3. La adresa *http://w3.one.net/~jhoffman/sqltut.htm* se găsește un bun curs introductiv despre SQL.
4. În unele SGBD-uri (de exemplu, PostgreSQL), numai o clauză este obligatorie.

```
FROM studenti
WHERE an = 3 AND specializare = 'Informatica Economica'
```

Comanda încearcă să extragă din tabela STUDENŢI toate liniile în care valoarea atributului an este *3* şi, în acelaşi timp, cea a atributului specializare este *Informatică Economică*. Pentru toate aceste linii extrase se afişează numai valoarea atributului NumePrenume.

Clauza SELECT corespunde deci operatorului proiecţie din algebra relaţională, fiind utilizată pentru desemnarea listei de atribute (coloanele) din rezultat. Clauza FROM este cea în care sunt enumerate tabelele din care vor fi extrase informaţiile aferente consultării. Cu WHERE se desemnează predicatul selectiv al algebrei relaţionale, relativ la atribute ale relaţiilor care apar în clauza FROM. Prin execuţia unei fraze SELECT se obţine un rezultat de formă tabelară. Acesta poate fi o tabelă:

- temporară anonimă (creată implicit la execuţia unei „fraze" SELECT ce nu are clauze speciale de stocare/direcţionare a rezultatului);
- „obişnuită":

  ```
  CREATE TABLE tab_stud_an3_InfoEc AS
  SELECT NumePrenume
  FROM studenti
  WHERE an = 3 AND specializare = 'Informatica Economica'
  ```

- temporară globală, cu structura stocată în dicţionarul de date şi conţinut limitat la o sesiune sau la o tranzacţie în care are loc invocarea tabelei:

  ```
  CREATE GLOBAL TEMPORARY TABLE gtt_stud_an3_InfoEc
  AS...
  ```

- temporară locală care este şi mai restrictivă decât tabela temporară globală în sensul că, dacă într-o sesiune/tranzacţie logica aplicaţiei presupune execuţia mai multor blocuri de cod, fiecare bloc ce face referire la tabela temporară locală va avea o instanţă proprie a tabelei, independentă de celelalte blocuri:

  ```
  CREATE LOCAL TEMPORARY TABLE ltt_stud_an3_InfoEc
  AS...
  ```

- virtuală[5]:

  ```
  CREATE VIEW view_stud_an3_InfoEc
  AS...
  ```

În aplicaţii, uneori rezultatul este obţinut şi ca o variabilă masivă (tablou) sau set de înregistrări (recordset) necesare populării unei liste, combo (elemente ale unui formular) etc.

Atunci când clauza WHERE este omisă, se consideră implicit că predicatul P are valoarea logică „adevărat", astfel încât vor fi incluse în rezultat toate liniile din tabelă sau produsul cartezian al tabelelor, enumerată/enumerate în clauza FROM. Dacă în locul coloanelor C1, C2, ..., Cn apare simbolul *, rezultatul va fi alcătuit din toate coloanele (atributele) relaţiilor specificate în clauza FROM. De asemenea,

5. Despre tabele temporare şi virtuale o să discutăm puţin şi în capitolul 14.

atributele rezultatului preiau numele din tabela/tabelele specificate în FROM. Schimbarea numelui se realizează prin clauza AS.

În capitolul 2 era subliniată echivalenţa noţiunilor de *relaţie* şi *tabelă*. Conform restricţiei de unicitate, este interzisă repetarea unei tuplu într-o relaţie. SQL-ul este tolerant în această privinţă. Spre deosebire de algebra relaţională, în SQL nu se elimină automat tuplurile identice (dublurile) din rezultat, ceea ce contravine uneia dintre poruncile fundamentale ale modelului relaţional. Aceasta este vestea proastă. Vestea bună este că se poate folosi opţiunea DISTINCT:

```
SELECT DISTINCT C1, C2, ..., Cn
FROM R1, R2, ..., Rm
WHERE P
```

În concluzie, o frază SELECT, în forma de mai sus, corespunde mai multor operatori algebrici relaţionali: selecţie (clauza WHERE – P); proiecţie (SELECT – Ci); produs cartezian (FROM – R1 ⊗ R2 ⊗ ... ⊗ Rm) şi conduce la obţinerea unui rezultat cu *n* coloane, fiecare coloană fiind un atribut din R1, R2, ..., Rm sau o expresie calculată pe baza unor atribute din R1, R2, ..., Rm.

Operatorii de bază folosiţi pentru comparaţiile dintre atribute/constante/expresii/funcţii din clauza WHERE sunt cei arhicunoscuţi:

- egal (=);
- diferit (<>);
- (strict) mai mic (<);
- mai mic sau egal (<=);
- (strict) mai mare (>);
- mai mare sau egal (>=).

Înainte de a trece la „conversia" în SQL a câtorva interogări din algebra relaţională, pe baza exemplelor din capitolul 4, mai trebuie făcute câteva precizări. Mai întâi, deşi nu e nicidecum obligatoriu, în cele mai multe interogări SQL vom scrie cuvintele rezervate (SELECT, FROM, WHERE, GROUP BY, HAVING, DISTINCT, SUM etc.) cu majuscule, numele tabelelelor cu litere mici, iar numele atributelor cum s-o nimeri. Vom încerca să ilustrăm cu figuri rezultatele a cât mai multor interogări. Însă capturile de ecran vor fi preponderent din Oracle şi PostgreSQL, ceea ce e oarecum nedrept pentru celelalte SGBD-uri luate în discuţie. Pentru cei care nu au avut prea des prilejul de a lucra cu servere de baze de date, trebuie spus că ordinea de afişare a liniilor în rezultatul unei interogări poate fi uneori derutantă. În lipsa clauzei ORDER BY, înregistrările sunt, iniţial, afişate în ordinea introducerii în tabele. După actualizări (DELETE-uri, urmate de INSERT-uri sau UPDATE-uri), ordinea poate să pară ciudată, întrucât SGBD-ul „reciclează" spaţiul eliberat la ştergerea unui tuplu[6]. Dar să revenim la exemplele din algebra relaţională.

---

6. Pentru detalii legate de organizarea fizică a înregistrărilor pe disc în Oracle, vezi şi [Fotache *et al.* 2003]. Lucrurile stau oarecum similar în mai toate serverele majore de baze de date, orice carte de administrare de baze de date explicând organizarea bazei la nivelele fizic şi logic.

Exemplul 1 – selecţie. Paragraful 4.4.1 (figura 4.7).

Interogarea SQL care obţine rezultatul din figura 4.7 este:

SELECT *

FROM r1

WHERE A > 20 AND C > 20

O variantă echivalentă este:

SELECT r1.* FROM r1 WHERE r1.A > 20 AND r1.C > 20

Din raţiuni de lizibilitate, de regulă, voi scrie fiecare clauză din comanda SELECT pe (cel puţin) o linie. Prefixarea atributelor (r1.A/r1.C/r1.*) nu este obligatorie în acest caz. Problema prefixării trebuie privită cu atenţie atunci când în clauza FROM apar două sau mai multe tabele ce prezintă atribute cu nume identice, după cum vom vedea nu peste mult timp.

Folosind utilitarul SQL Developer distribuit gratuit de către cei de la Oracle, dialogul poate arăta ca în figura 5.1. Conexiunea dintre client şi serverul de date a fost denumită SQL2008, iar rezultatul interogării are liniile numerotate (în exemplul nostru e o singură linie, aşa încât nu prea a fost de lucru cu numerotarea).

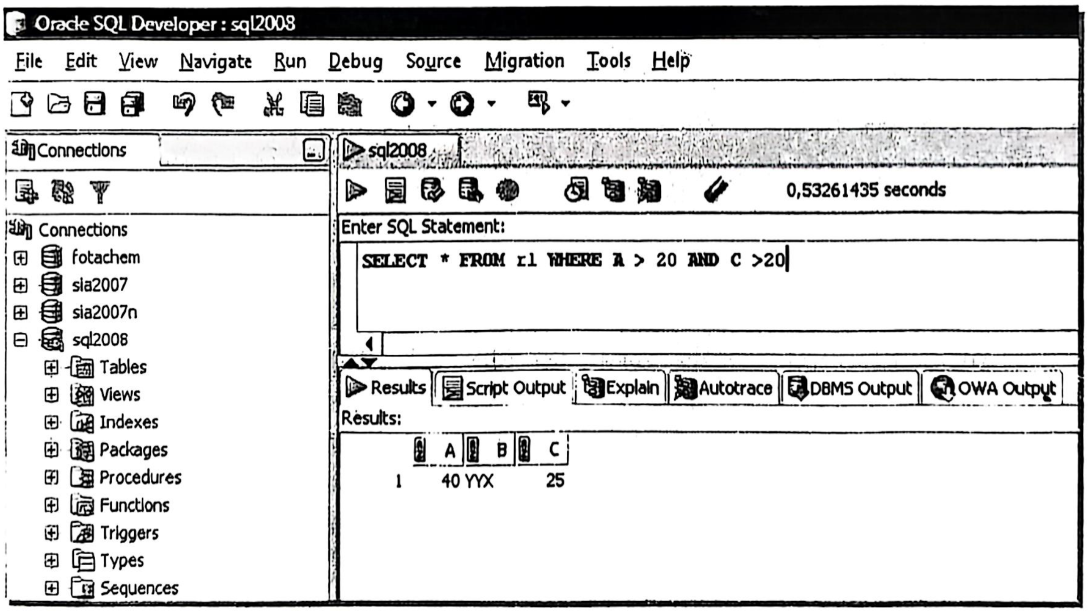


Figura 5.1. Interogare simplă executată în Oracle SQL Developer

Exemplul 2 – selecţie. Care sunt judeţele din Moldova? – paragraful 4.4.1 (figura 4.8).

Interogarea următoare funcţionează fără modificări în toate SGBD-urile care pretind că au implementat SQL:

SELECT *

FROM judete

WHERE Regiune = 'Moldova'

După cum am văzut în capitolele precedente, utilitarul standard de conectare a unui client la serverul PostgreSQL-ul este pgAdmin. Execuția interogării de mai sus în pgAdmin se poate solda cu un rezultat de forma celui din figura 5.2.

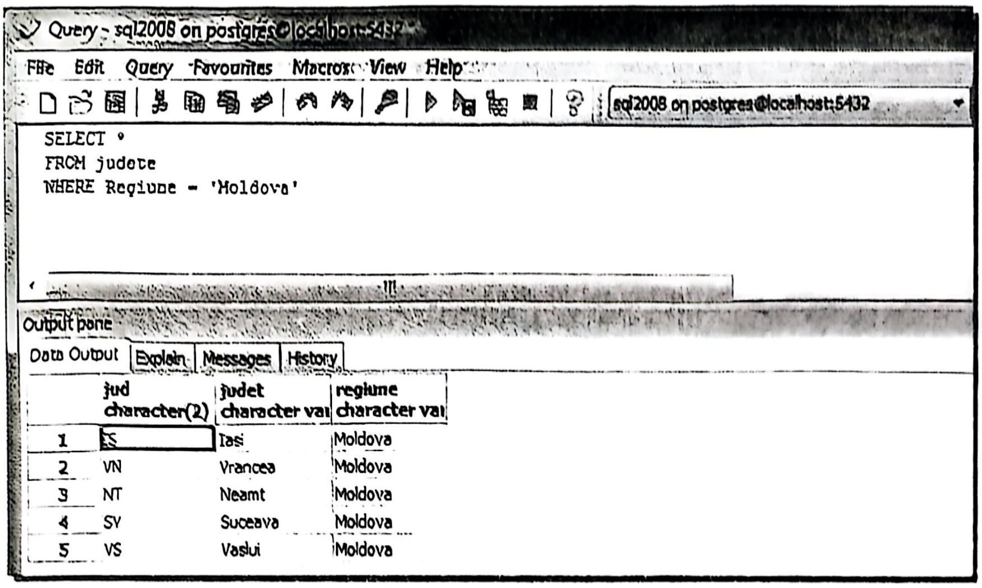


Figura 5.2. Interogare executată în pgAdmin (PostgreSQL)

Mai toate interfețele serverelor BD afişează un număr pentru fiecare linie din rezultat. Unele, precum Oracle SQL Developer sau Microsoft SQL Server Management Studio (vezi figura 5.3), afişează explicit valorile NULL.

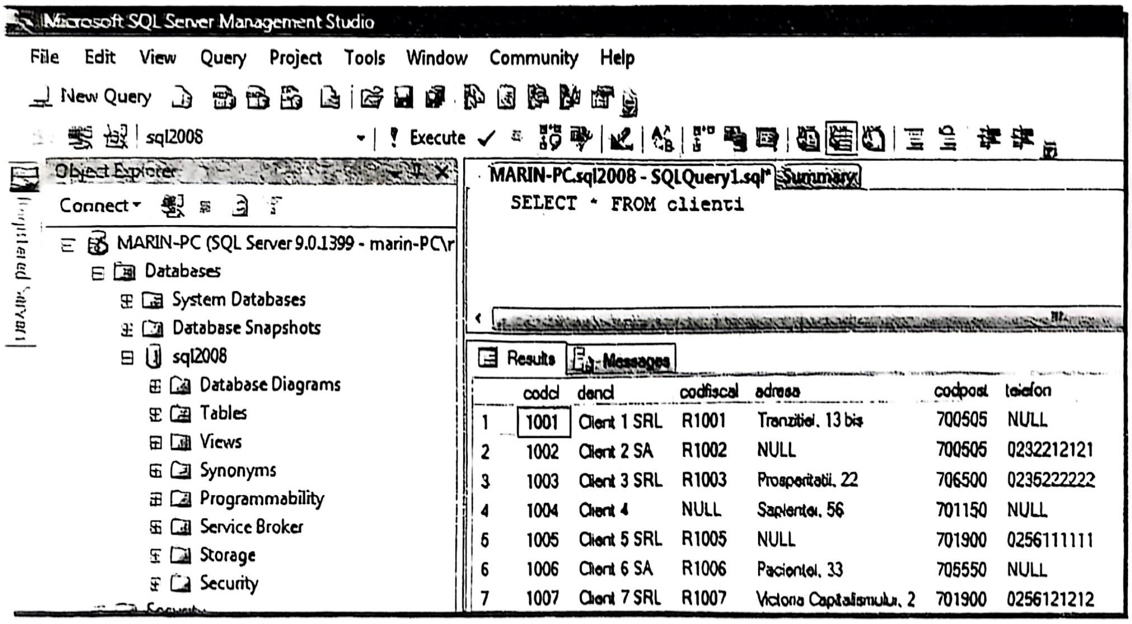
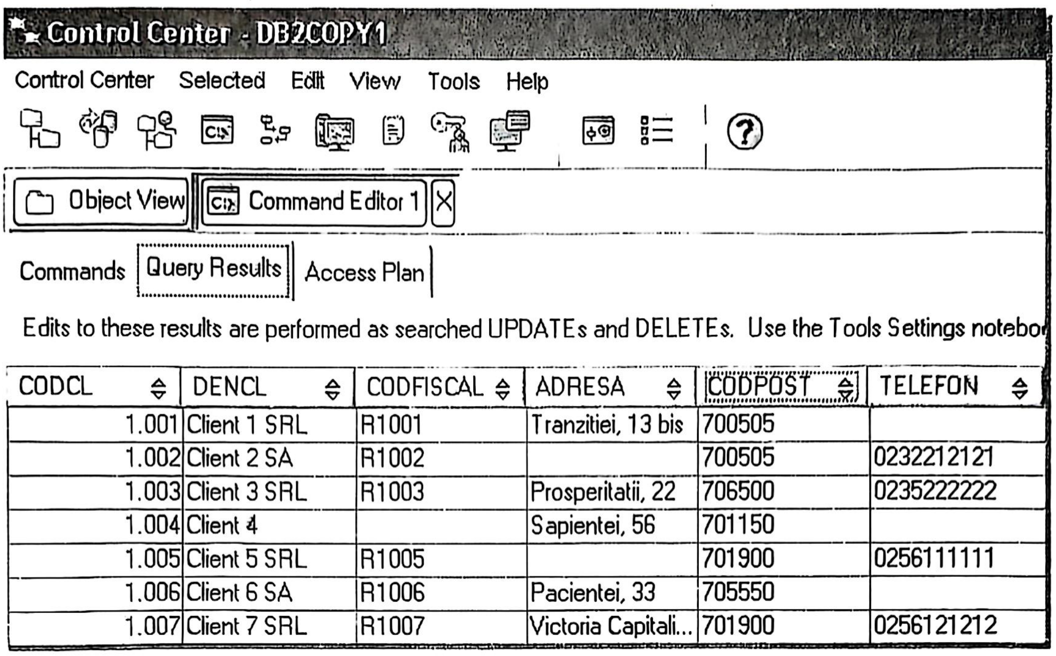


Figura 5.3. Afişarea explicită a valorilor NULL (cazul Microsoft SQL Server)

Altele, precum PostgreSQL sau DB2 lasă spații libere (vezi figura 5.4), astfel încât este destul de greu să ne dăm seama unde este un NULL sau unul sau mai multe spații.



CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1.001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	
1.002	Client 2 SA	R1002		700505	0232212121
1.003	Client 3 SRL	R1003	Prosperitatii, 22	706500	0235222222
1.004	Client 4		Sapientei, 56	701150	
1.005	Client 5 SRL	R1005		701900	0256111111
1.006	Client 6 SA	R1006	Pacientei, 33	705550	
1.007	Client 7 SRL	R1007	Victoria Capitali...	701900	0256121212

Figura 5.4. Afişarea defectuoasă (de fapt, neafişarea) a valorilor NULL (cazul DB2 Control Center)

Exemplul 3 - selecţie. Care sunt facturile emise în perioada 2-5 august 2007? - paragraful 4.4.1 (figura 4.9).

Formatul standard al unei constante de tip dată calendaristică este *DATE'YYYY-MM-DD'*, aşa încât interogarea se prezintă astfel:

```
SELECT *
FROM facturi
WHERE DataFact >= DATE'2007-08-02' AND DataFact <= DATE'2007-08-05'
```

În unele SGBD-uri, însă (de exemplu, SQL Server), acest format de dată nu este recunoscut implicit, fiind necesară configurarea sesiunii curente pe formatul dorit sau utilizarea unor clauze sau funcţii suplimentare. Altele, precum Oracle sau PostgreSQL, acceptă standardul *DATE'YYYY-MM-DD'*, dar folosesc, de multe versiuni încoace, funcţii proprii, precum TO_DATE:

```
SELECT *
FROM facturi
WHERE DataFact >= TO_DATE('02/08/2007', 'DD/MM/YYYY') AND
DataFact <= TO_DATE('05/08/2007', 'DD/MM/YYYY')
```

În DB2 şi SQL Server, niciunul dintre formatele de mai sus nu este recunoscut, însă acest lucru nu este prea deranjant, întrucât literalii (constantele) de tip dată calendaristică se încadrează între apostrofuri pe formatul *'YYYY-MM-DD'*:

```
SELECT *
FROM facturi
WHERE DataFact >= '2007-08-02' AND DataFact <= '2007-08-05'
```

Cel mai generos este PostgreSQL, în care funcţionează inclusiv această ultimă versiune a interogării.

Exemplul 4 - proiecție. Paragraful 4.4.2 (figura 4.10).

SELECT A,C

FROM r1

Cele trei linii ale rezultatului din figura 4.10 se obțin întocmai și în SQL, întrucât valorile combinației (A,C) nu se repetă în R1.

Exemplul 5. Care sunt regiunile țării preluate în bază? - paragraful 4.4.2 (figura 4.11).

Acum apare o primă diferențiere de rezultatul din algebra relațională. După cum aminteam cu câteva zeci ce rânduri mai sus, spre deosebire de algebra relațională, SQL nu elimină automat dublurile. Tabela obținută prin consultarea:

SELECT Regiune

FROM judete

are structura și conținutul identice cu R' din figura 4.11. Pentru a obține răspunsul de forma tabelei R (o regiune să apară o singură dată în răspuns) se folosește clauza DISTINCT:

SELECT DISTINCT Regiune

FROM judete

Exemplul 6. Care sunt: codul, denumirea și numărul de telefon ale fiecărui client? - paragraful 4.4.2 (figura 4.12).

SELECT CodCl, DenCl, Telefon

FROM clienti

Nu este necesară clauza DISTINCT, deoarece CodCl este cheia primară a tabelei CLIENȚI.

Exemplul 7. Care este numărul de telefon al clientul Client 2 SA? - paragraful 4.4.3 (figura 4.13).

Dacă în algebra relațională aveam nevoie de înlănțuirea a doi operatori, SQL-ul este mai generos, dat fiind formatul comenzii SELECT:

SELECT Telefon

FROM clienti

WHERE DenCl = 'Client 2 SA'

Exemplul 8. Care sunt denumirile și codurile poștale ale localităților (prezente în bază) din județele Iași (IS) și Vrancea (VN)? - paragraful 4.4.3 (figura 4.14).

SELECT Loc, CodPost

FROM coduri_postale

WHERE Jud = 'IS' OR Jud = 'VN'

Paragraful 5.3 va fi dedicat coloanelor calculate pe baza unor expresii în care sunt implicate atribute din tabele în combinație cu diverse constante (literali) și

funcţii. Până atunci însă, ne întrebăm: ce se afişează la execuţia următoarei interogări?[7]

SELECT CURRENT_DATE FROM judete

CURRENT_DATE este funcţia SQL prin care se returnează data sistemului (calculatorului). Majoritatea dialectelor SQL au clauza FROM obligatorie într-o comandă SELECT. Eu am scris, aproape la întâmplare, în clauza FROM una dintre tabelele bazei de date. Cum procedează SGBD-ul? Simplu: 1) se verifică existenţa tabelei JUDEŢE; 2) se extrag din tabela JUDEŢE numai liniile care satisfac condiţia din clauza WHERE; aici clauza WHERE lipseşte, deci se extrag toate liniile tabelei; 3) pentru toate liniile extrase la punctul anterior, se afişează coloanele din clauza SELECT; chiar dacă pare ciudat, SQL-ul nu se supără că în clauza SELECT nu apare niciun atribut din JUDEŢE. Pur şi simplu, se va afişa data curentă pe un număr de rânduri egal cu numărul liniilor tabelei - vezi figura 5.5 (inutil de amintit că această tulburătoare interogare a fost executată ultima oară pe 10 octombrie 2008).

CURRENT_DATE
10-OCT-08
10-OCT-08
10-OCT-08
10-OCT-08
10-OCT-08
10-OCT-08

Figura 5.5. Interogare ce afişează (repetat) data curentă

Ca să reducem numărul liniilor din rezultat la unu, am putea folosi clauza DISTINCT:

SELECT DISTINCT CURRENT_DATE FROM judete

Deşi funcţională, soluţia este destul de stupidă. Motiv pentru care diverşi producători de SGBD-uri s-au gândit la variante mai acceptabile. Spre exemplu, în PostgreSQL este posibilă renunţarea la clauza FROM, interogarea:

SELECT CURRENT_DATE

fiind perfect funcţională. Microsoft SQL Server nu are implementată funcţia CURRENT_DATE, ci numai CURRENT_TIMESTAMP, iar, ca şi în PostgreSQL, clauza FROM poate lipsi:

SELECT GETDATE()

sau

SELECT CURRENT_TIMESTAMP

Alte SGBD-uri, însă, au recurs la soluţia creării automate (la instalare) a unei tabele cu o sigură linie şi un singur atribut, astfel încât, în lipsa clauzei WHERE rezultatul va conţine, cu siguranţă, un singur rând. În Oracle, această tabelă se numeşte DUAL, iar în DB2 SYSIBM.SYSDUMMY1, interogarea fiind în Oracle:

7. În Microsoft SQL Server, varianta executabilă este: SELECT GETDATE() FROM judete.

```
SELECT CURRENT_DATE FROM dual
```

iar în DB2:

```
SELECT CURRENT_DATE FROM SYSIBM.SYSDUMMY1
```

De la versiunea 9.5, tabela DUAL există şi în DB2 (în schema SYSIBM), aşa că este corectă şi sintaxa:

```
SELECT CURRENT_DATE FROM SYSIBM.dual
```

5.2. Reuniuni, intersecţii, diferenţe, produse carteziene

Cei patru operatori ansamblişti din algebra relaţională au fost transpuşi în SQL într-o sintaxă destul de uşor de înţeles. Apar, totuşi, câteva discuţii legate de titulatura unor operatori în diverse dialecte, plus eterna problemă a liniilor duplicat.

5.2.1. Reuniunea

Un rezultat identic cu tabela R3 din figura 4.3 (paragraful 4.3.1) se obţine prin următoarea frază SELECT:

```
SELECT *
FROM r1
        UNION
SELECT *
FROM r2
```

La reuniunea a două tabele, SQL elimină automat liniile identice din rezultat. Dacă se doreşte preluarea tuturor liniilor celor două relaţii şi, implicit, apariţia liniilor duplicat (ceea ce, reamintim, contrazice un principiu de bază al modelului relaţional) se foloseşte clauza ALL astfel:

```
SELECT *
FROM r1
        UNION ALL
SELECT *
FROM r2
```

Rezultatul (vezi figura 5.6) conţine câte o linie pentru fiecare tuplu din cele două tabele.

A	B	C
20	XYZ	30
30	XXZ	20
40	YYX	25
25	XYZ	30
40	YYX	25
30	XXZ	40

Figura 5.6. Rezultatul operatorului UNION ALL

Revenim la exemplul 8 (Care sunt denumirile şi codurile poştale ale localităţilor (prezente în bază) din judeţele Iaşi (IS) şi Vrancea (VN)?) - paragraful 4.4.3 (figura 4.14) pe care l-am discutat cu două pagini în urmă. Dacă soluţia precedentă se baza pe un predicat ce folosea operatorul logic *sau* (OR), acum suntem în măsură să formulăm şi o soluţie bazată pe reuniune:

```
SELECT Loc, CodPost
FROM coduri_postale
WHERE Jud = 'IS'
        UNION
SELECT Loc, CodPost
FROM coduri_postale
WHERE Jud = 'VN'
```

5.2.2. Intersecţia

Raportându-ne la exemplul din paragraful 4.3.2 (figura 4.4), echivalentul tabelei R4 se obţine în SQL prin:

```
SELECT *
FROM r1
        INTERSECT
SELECT *
FROM r2
```

Stardardele SQL permit, ca şi în cazul reuniunii, prezenţa repetată în rezultat a unor linii (tupluri duplicat), bineînţeles atunci când cele două relaţii asupra cărora se aplică intersecţia nu respectă restricţia de unicitate:

```
SELECT *
FROM r1
        INTERSECT ALL
SELECT *
FROM r2
```

Dacă tuplul t1 apare repetat şi în R1, şi în R2, mai precis, de *n* ori în R1 şi de *m* ori în R2, în rezultatul operatorului INTERSECT t1 apare o singură dată, în timp ce utilizând INTERSECT ALL t1 va apărea de un număr de ori care este minimul dintre *n* şi *m*. INTERSECT ALL nu funcţionează nici în Oracle, nici în MS SQL Server, în timp ce în DB2 şi PostgreSQL INTERSECT ALL furnizează acelaşi rezultat ca INTERSECT!

Pentru o primă ilustrare a utilizării intersecţiei, transcriem soluţia din algebra relaţională formulată în paragraful 4.3.3, exemplul 9 (vezi figura 4.16, Care sunt codurile produselor ce apar simultan şi în factura 1111, şi în factura 1117?):

```
SELECT CodPr
FROM liniifact
WHERE NrFact = 1111
```

```
        INTERSECT
SELECT CodPr
FROM liniifact
WHERE NrFact = 1117
```

Exemplele de mai sus funcționează în DB2, Oracle şi PostgreSQL, nu însă şi în alte SGBD-uri (de exemplu, Visual FoxPro sau Access) care nu are implementat operatorul INTERSECT, astfel încât intersecția trebuie realizată prin alte clauze şi operatori.

5.2.3. Diferența

Operatorul aşteptat ar fi MINUS, titulatură sub care a fost implementat în Oracle. În standardele SQL şi în alte câteva SGBDR-uri (precum DB2), titulatura operatorului pentru diferență (scădere relațională) nu e MINUS, ci EXCEPT, iar în alte SGBD-uri (Visual FoxPro, Access) nu există nici unul, nici altul. Tabela R5 din figura 4.5 (paragraful 4.3.3), calculată prin expresia R1 - R2, se obține în SQL (şi PostgreSQL/DB2/MS SQL Server) prin interogarea:

```
SELECT *
FROM r1
        EXCEPT
SELECT *
FROM r2
```

Ca şi în cazul reuniunii şi intersecției, eliminarea tuplurilor duplicat se face automat, iar repetarea lor se asigură prin EXCEPT ALL. În DB2 şi PostgreSQL, dacă INTERSECT ALL avea o funcționalitate identică INTERSECT-ului, EXCEPT ALL funcționează „corect". În Oracle, MINUS ALL nu este implementat, ca şi EXCEPT ALL în SQL Server.

Modificăm enunțul ultimului exemplu din paragraful precedent: Care sunt codurile produselor care apar în factura 1111, dar nu apar în factura 1117 ?

```
SELECT CodPr
FROM liniifact
WHERE NrFact = 1111
        EXCEPT
SELECT CodPr
FROM liniifact
WHERE NrFact = 1117
```

Execuția acestei consultări (în PostgreSQL) se soldează cu rezultatul din figura 5.7.

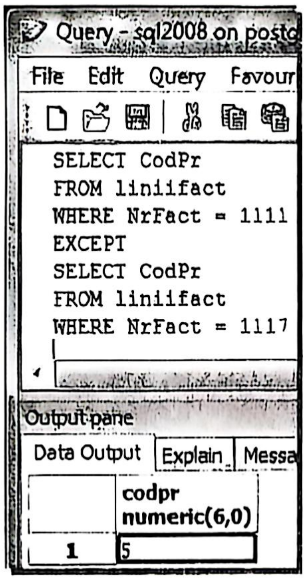


Figura 5.7. Codul produsului care apare în factura 1111, dar nu apare în 1117 (execuţie în PostgreSQL pgAdmin)

Care sunt facturile care nu sunt încasate nici măcar parţial?

Practic, ne interesează facturile ce nu prezintă nicio tranşă de încasare. Dacă tabela FACTURI conţine facturile emise, tranşele de facturi încasate se află în tabela INCASFACT. Rezultatul acestei probleme (figura 5.8) poate fi aflat prin diferenţa dintre facturile emise şi cele cu încasări. Asigurarea unicompatibilităţii rezultatelor celor două fraze SELECT conectate prin EXCEPT va fi asigurată prin selectarea unui singur atribut - NrFact:

```
SELECT DISTINCT NrFact
FROM liniifact
        EXCEPT
SELECT NrFact
FROM incasfact
```

NrFact
1114
1115
1116
1119
1121
1122
2111
2112
2113
2115
2116
2117
2118
2119
2121
2122
3111
3112
3113
3115
3116
3117
3118
3119

Figura 5.8. Facturile fără nicio încasare

5.2.4. Produsul cartezian

Standardele SQL-89 şi SQL-92 nu puneau la dispoziţie vreun operator special dedicat produsului cartezian. Nici nu era mare nevoie. Tabela R6 din figura 4.6 (paragraful 4.3.4) se poate obţine, pur şi simplu, prin enumerarea celor două relaţii în clauza FROM:

```
SELECT *
FROM r1, r2
```

Din SQL:1999 apare însă operatorul CROSS JOIN implementat în toate cele patru servere de baze de date:

```
SELECT *
FROM r1 CROSS JOIN r2
```

5.3. Coloane-expresii

O facilitate importantă în multe interogări SQL ţine de definirea, pe lângă atributele tabelelor, a unor coloane noi, pe baza unor expresii. Despre funcţiile aplicate atributelor sau constantelor şi altor funcţii vom discuta în capitolul următor. Clauza AS permite denumirea coloanelor calculate sau redenumirea unor coloane ale tabelelor. Dacă numele coloanei conţine spaţii sau alte caractere neautorizate, trebuie încadrat între ghilimele (nu între apostrofuri, precum literalii şir de caractere)[8]. Să luăm un exemplu banal: *Care este, pentru fiecare produs din factura 1111, codul, cantitatea, preţul unitar şi valoarea fără TVA?*

```
SELECT CodPr AS "Cod produs",
Cantitate AS "Cantitate",
PretUnit AS "Pret Unitar",
        Cantitate * PretUnit AS "Valoare fara TVA"
FROM liniifact
WHERE NrFact = 1111
```

Rezultatul interogării este prezentat în figura 5.9.

Cod produs	Cantitate	Pret Unitar	Valoare fara TVA
1	50	1000	50000
2	75	1050	78750
5	500	7060	3530000

Figura 5.9. Exemplu de coloană calculată

A patra coloana este denumită *Valoare fără TVA*, după cum a fost specificat în clauza AS. Valorile sale sunt determinate pe baza expresiei *Cantitate * Preţ Unitar*.

8. În SQL Server, în locul ghilimelelor se pot folosi parantezele drepte (de exemplu, [Preţ Unitar]).

Acum că am arătat cum se poate modifica numele unei coloane, vom reveni la variantă mai comodă şi vom folosi mai rar spaţii sau alte caractere (cu excepţia underscore) pentru a evita utilizarea ghilimelelor.

Atenţie, însă! Dacă numele definit prin clauza AS poate fi folosit la ordonare, includerea sa în predicatul din clauza WHERE se soldează cu o eroare. Spre exemplu, dintre liniile facturii 1111 ne interesează doar cele în care valoarea fără TVA este mai mare decât 60000. Interogarea:

```
SELECT CodPr AS "Cod produs", Cantitate AS "Cantitate",
PretUnit AS "Pret Unitar", Cantitate * PretUnit AS "Valoare fara TVA"
FROM liniifact
WHERE NrFact = 1111 AND "Valoare fara TVA" > 60000
```

se va solda cu un mesaj de eroare - vezi figura 5.10.

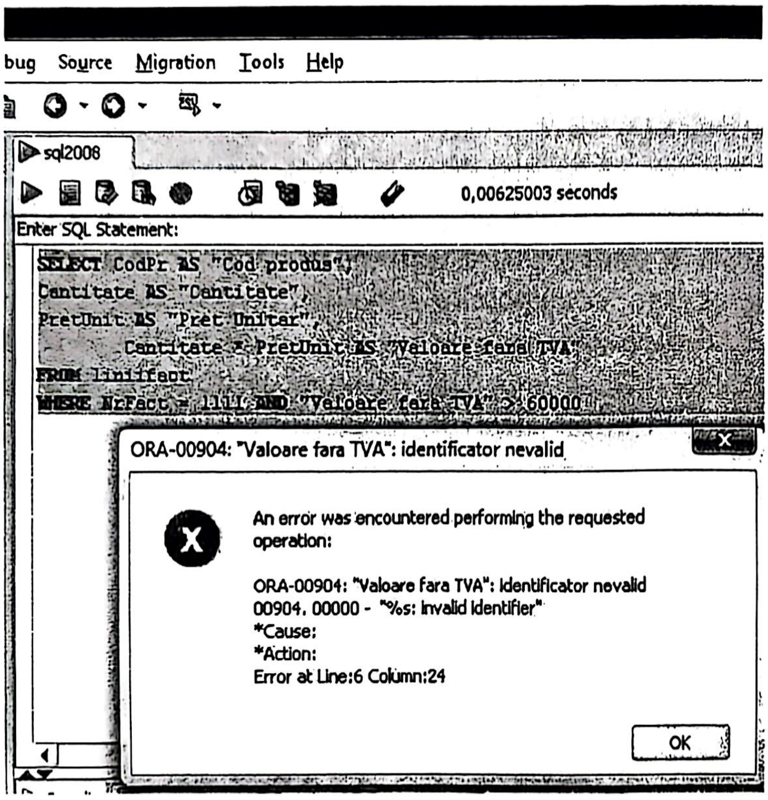


Figura 5.10. Eroare generată de folosirea numelui unei coloane calculate în clauza WHERE

Versiunea corectă presupune includerea în predicatul de selecţie nu a numelui, ci a expresiei ce defineşte coloana respectivă:

```
SELECT CodPr AS "Cod produs",
       Cantitate AS "Cantitate",
       PretUnit AS "Pret Unitar",
       Cantitate * PretUnit AS "Valoare fara TVA"
FROM liniifact
WHERE NrFact = 1111 AND Cantitate * PretUnit > 60000
```

Un alt tip de expresie este cel bazat pe concatenare, adică pe alipirea mai multor constante, variabile şi atribute într-o coloană nouă. Operatorul SQL pentru

concatenare este alcătuit din două bare verticale (| |). Spre exemplu, interogarea următoare produce rezultatul din figura 5.11[9].

```
SELECT 'Judetul ' || Judet || ' se afla in ' || Regiune
        AS Exemplu_Concatenare
FROM judete
```

EXEMPLU_CONCATENARE
Judetul Iasi se afla in Moldova
Judetul Vrancea se afla in Moldova
Judetul Neamt se afla in Moldova
Judetul Suceava se afla in Moldova
Judetul Vaslui se afla in Moldova
Judetul Timis se afla in Banat

Figura 5.11. Concatenarea unor literali şi atribute de tip şir de caractere

SQL este un limbaj puternic tipizat, în sensul că pot fi combinaţi în expresii numai operatori de acelaşi tip. Conversia unui operator dintr-un tip în altul (acolo unde este posibil) se face cu ajutorul funcţiei CAST. În unele SGBD-uri, însă (de exemplu, PostgreSQL şi Oracle), există o mai mare libertate. Astfel, se pot concatena direct, adică fără conversie prealabilă, literali, atribute-şir de caractere, numerice etc. Alte SGBD-uri, precum DB2, necesită transformarea valorilor non-şir de caractere (atribute/constante numerice, dată calendaristică etc.), operaţiune realizată prin funcţia CAST. Spre exemplu, în Oracle şi PostgreSQL interogarea următoare este funcţională:

```
SELECT 'Factura ' || NrFact || ' a fost emisa pe data ' ||
        DataFact AS Concatenare_Oracle_PgSQL
FROM facturi
```

în timp ce DB2 nu o acceptă. Folosind funcţia de conversie CAST, putem redacta următoarea variantă a interogării care funcţionează în DB2, dar şi în PostgreSQL şi Oracle:

```
SELECT 'Factura ' || CAST (NrFact AS CHAR(8)) ||
        ' a fost emisa pe data ' || CAST (DataFact AS VARCHAR(10))
        AS Concatenare_DB2_PgSQL
FROM facturi
```

Valorile atributului NrFact sunt transformate în şiruri de caractere de lungime fixă (8 poziţii), în timp ce ale DataFact vor fi convertite în şiruri de caractere de lungime variabilă (vorba vine, deoarece formatul datei este unitar pentru toate liniile rezultatului) de maximum 10 poziţii. După cum se observă în figura 5.12, dacă PostgreSQL converteşte valoarea NrFact pe un şir de caractere de lungimea numărului de cifre, în DB2 la conversie se completează cu zerouri la stânga până la completarea celor opt poziţii (*CAST NrFact AS VARCHAR(8)*).

9. În MS SQL Server, pentru concatenare se foloseşte simbolul +.

concatenare_pgsql text
Factura 1111 a fost emisa pe data 2007-08-01
Factura 1112 a fost emisa pe data 2007-08-01
Factura 1113 a fost emisa pe data 2007-08-01
Factura 1114 a fost emisa pe data 2007-08-01
Factura 1115 a fost emisa pe data 2007-08-02
Factura 1116 a fost emisa pe data 2007-08-02
Factura 1117 a fost emisa pe data 2007-08-03
Factura 1118 a fost emisa pe data 2007-08-04
Factura 1119 a fost emisa pe data 2007-08-07
Factura 1120 a fost emisa pe data 2007-08-07
Factura 1121 a fost emisa pe data 2007-08-07
Factura 1122 a fost emisa pe data 2007-08-07
Factura 2111 a fost emisa pe data 2007-08-14
Factura 2112 a fost emisa pe data 2007-08-14
Factura 2113 a fost emisa pe data 2007-08-14
Factura 2115 a fost emisa pe data 2007-08-15

CONCATENARE_DB2
Factura 00001111 a fost emisa pe data 2007-08-01
Factura 00001112 a fost emisa pe data 2007-08-01
Factura 00001113 a fost emisa pe data 2007-08-01
Factura 00001114 a fost emisa pe data 2007-08-01
Factura 00001115 a fost emisa pe data 2007-08-02
Factura 00001116 a fost emisa pe data 2007-08-02
Factura 00001117 a fost emisa pe data 2007-08-03
Factura 00001118 a fost emisa pe data 2007-08-04
Factura 00001119 a fost emisa pe data 2007-08-07
Factura 00001120 a fost emisa pe data 2007-08-07
Factura 00001121 a fost emisa pe data 2007-08-07
Factura 00001122 a fost emisa pe data 2007-08-07
Factura 00002111 a fost emisa pe data 2007-08-14
Factura 00002112 a fost emisa pe data 2007-08-14
Factura 00002113 a fost emisa pe data 2007-08-14
Factura 00002115 a fost emisa pe data 2007-08-15
Factura 00002116 a fost emisa pe data 2007-08-16

Figura 5.12. O diferenţă la conversia număr – şir de caractere între PostgreSQL şi DB2

SQL Server foloseşte semnul plus pentru concatenare şi, similar DB2-ului, reclamă transformarea explicită a argumentelor în şiruri de caractere:

```
SELECT 'Factura ' + CAST (NrFact AS CHAR(8)) +
        ' a fost emisa pe data ' + CAST (DataFact AS VARCHAR(11))
        AS Concatenare_SQLServer
FROM facturi
```

În finalul discuţiei despre coloanele-expresii, mai zăbovim preţ de câteva rânduri la expresiile de tip dată calendaristică. Modurile în care au fost implementate aceste funcţii sunt foarte eterogene de la SGBD la SGBD. Să presupunem că orice factură trebuie încasată în maximum două săptămâni de la data emiterii. DataFact este un atribut de tip DATE, aşa că numărul 14 din expresia ce calculează scadenţa este „tratat" ca număr de zile în toate cele patru SGBD-uri, cu excepţia DB2:

```
SELECT NrFact AS Factura,
        DataFact AS Data_Facturare,
        DataFact + 14 AS Scadenta_Incasare
FROM facturi
```

Specificarea intervalelor calendaristice este o sarcină relativ uşoară. Astfel, interogarea următoare este redactată în PostgreSQL şi furnizează acelaşi rezultat (vezi figura 5.13) în privinţa datei scadente:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + 14 AS Scadenta_Incasare1,
        DataFact + INTERVAL '14 DAYS' AS Scadenta_Incasare2,
        DataFact + INTERVAL '2 WEEKS' AS Scadenta_Incasare3
FROM facturi
```

factura numeric(8,0)	data_factura date	scadenta_incasare1 date	scadenta_incasare2 timestamp without time zone	scadenta_incasare3 timestamp without time zone
1111	2007-08-01	2007-08-15	2007-08-15 00:00:00	2007-08-15 00:00:00
1112	2007-08-01	2007-08-15	2007-08-15 00:00:00	2007-08-15 00:00:00
1113	2007-08-01	2007-08-15	2007-08-15 00:00:00	2007-08-15 00:00:00
1114	2007-08-01	2007-08-15	2007-08-15 00:00:00	2007-08-15 00:00:00
1115	2007-08-02	2007-08-16	2007-08-16 00:00:00	2007-08-16 00:00:00
1116	2007-08-02	2007-08-16	2007-08-16 00:00:00	2007-08-16 00:00:00
1117	2007-08-03	2007-08-17	2007-08-17 00:00:00	2007-08-17 00:00:00
1118	2007-08-04	2007-08-18	2007-08-18 00:00:00	2007-08-18 00:00:00
1119	2007-08-07	2007-08-21	2007-08-21 00:00:00	2007-08-21 00:00:00
1120	2007-08-07	2007-08-21	2007-08-21 00:00:00	2007-08-21 00:00:00
1121	2007-08-07	2007-08-21	2007-08-21 00:00:00	2007-08-21 00:00:00
1122	2007-08-07	2007-08-21	2007-08-21 00:00:00	2007-08-21 00:00:00

Figura 5.13. De trei ori scadență (primele linii din rezultat)

DB2 este mai pretențios, nepermițând nici expresia de la *Scadență_Încasare1*, nici de la *Scadență_Încasare3* (din cauza clauzei WEEKS), ci doar:

```
SELECT NrFact AS Factura,
        DataFact AS Data_Facturare,
        DataFact + 14 DAYS AS Scadenta_Incasare2
FROM facturi
```

În Oracle, versiunea funcțională a interogării PostgreSQL de mai sus este:

```
SELECT NrFact AS Factura,
        DataFact AS Data_Facturare,
        DataFact + 14 AS Scadenta_Incasare1,
        DataFact + INTERVAL '14' DAY AS Scadenta_Incasare2
FROM facturi
```

intervalele de tip săptămâni (WEEK) nefiind implementate. Observați diferența la specificarea intervalului în zile. Aceeași interogare executată în PostgreSQL ne dă prilejul descoperirii unei ciudățenii a produsului - vezi figura 5.14. Deși nu ni se „spune" că am fi greșit undeva la sintaxă, a doua coloană-scadență (Scadență_Încasare2) are valori greșite - de fapt, valorile datei întocmirii fiecărei facturi. Din păcate, trebuie să avem grijă tot timpul, de aici încolo, ca în PostgreSQL la calificarea intervalelor să se folosească notația INTERVAL '*n* DAYS | WEEKS...'.

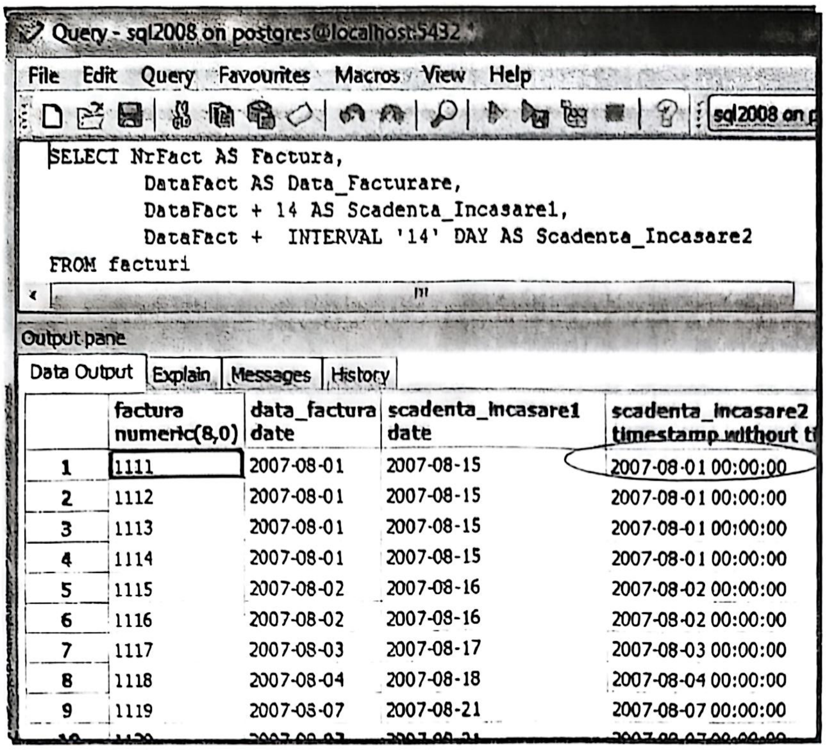


Figura 5.14. Un prim ghimpe în PostgreSQL

Dacă am presupune că scadenţa ar fi peste două luni de la facturare, interogarea ar trebui modificată astfel în PostgreSQL:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '2 MONTHS' AS Scadenta_Incasare
FROM facturi
```

Varianta următoare funcţionează şi în Oracle, şi în PostgreSQL, dar rezultatul corect este furnizat numai în Oracle.

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '2' MONTH AS Scadenta_Incasare
FROM facturi
```

Din păcate, SQL Server-ul de la Microsoft nu are implementat tipul INTERVAL şi nici clauzele de mai sus, operaţiunile de adunare şi scădere între date şi intervale calendaristice fiind realizate cu funcţii specifice precum DATEADD, DATEDIFF, CONVERT/CAST, după cum vom vedea în capitolul următor.

Complicându-ne şi mai zdravăn, *dorim să afişăm pentru fiecare factură ce dată va fi peste 1 an, două luni şi 25 de zile de la momentul emiterii.*

Soluţia 1 PostgreSQL:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '1 YEAR' +
        INTERVAL '2 MONTH' + INTERVAL '25 DAY'
        AS O_Data_Viitoare
FROM facturi
```

Soluţia 2 PostgreSQL:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '1 YEAR 2 MONTH 25 DAY' AS O_Data_Viitoare
FROM facturi
```

Soluția Oracle:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '1-2' YEAR TO MONTH +
        25 AS O_Data_Viitoare
FROM facturi
```

Soluția DB2:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + 1 YEAR + 2 MONTHS + 25 DAYS AS O_Data_Viitoare
FROM facturi
```

În ceea ce privește operațiunile de scădere a două date calendaristice, de regulă rezultatul se exprimă în zile. Spre exemplu, dacă ne interesează *numărul de zile scurse între 22 noiembrie 2007 și data emiterii fiecărei facturi* (figura 5.15 conține doar o parte dintre facturi), soluția Oracle/PostgreSQL este una simplă:

```
SELECT NrFact, DATE'2007-11-22', DataFact,
        DATE'2007-11-22' - DataFact AS Zile_Scurse
FROM facturi
```

NRFACT	DATE'2007-11-22'	DATAFACT	ZILE_SCURSE
1111	22-11-2007	01-08-2007	113
1112	22-11-2007	01-08-2007	113
1113	22-11-2007	01-08-2007	113
1114	22-11-2007	01-08-2007	113
1115	22-11-2007	02-08-2007	112
1116	22-11-2007	02-08-2007	112
1117	22-11-2007	03-08-2007	111
1118	22-11-2007	04-08-2007	110
1119	22-11-2007	07-08-2007	107
1120	22-11-2007	07-08-2007	107
1121	22-11-2007	07-08-2007	107
1122	22-11-2007	07-08-2007	107

Figura 5.15. Diferența în zile dintre două date calendaristice

Dacă dorim același rezultat, dar exprimat în ani și luni, operațiunea era destul de complicată până la standardul SQL:1999. Introducându-se tipurile de date INTERVAL, conversia se face într-o manieră simplă, scriind imediat după expresia-scădere tipul intervalului (YEAR TO MONTH, în cazul Oracle):

```
SELECT NrFact, DATE'2007-11-22', DataFact,
        DATE'2007-11-22' - DataFact AS Zile_Scurse,
        (DATE'2007-11-22' - DataFact) YEAR TO MONTH AS AniLuni_Sc
FROM facturi
```

Rezultatul Oracle (primele linii) este cel din figura 5.16.

NRFACT	DATE'2007-11-22'	DATAFACT	ZILE_SCURSE	ANILUNI_SC
1111	22-11-2007	01-08-2007	113	0-4
1112	22-11-2007	01-08-2007	113	0-4
1113	22-11-2007	01-08-2007	113	0-4
1114	22-11-2007	01-08-2007	113	0-4
1115	22-11-2007	02-08-2007	112	0-4
1116	22-11-2007	02-08-2007	112	0-4
1117	22-11-2007	03-08-2007	111	0-4
1118	22-11-2007	04-08-2007	110	0-4
1119	22-11-2007	07-08-2007	107	0-4
1120	22-11-2007	07-08-2007	107	0-4
1121	22-11-2007	07-08-2007	107	0-4
1122	22-11-2007	07-08-2007	107	0-4

Figura 5.16. Diferenţa în zile, ani şi luni dintre două date calendaristice (Oracle)

Interogarea funcţionează în Oracle, nu însă şi în PostgreSQL. După cum vom vedea însă în paragraful 6.4, există variante elegante bazate pe funcţia AGE. În DB2, diferenţa dintre două date calendaristice are aparenţa unui număr (de fapt, tipul este DURATION), care reprezintă anii, lunile şi zilele scurse dintre cele două date. Astfel, în urma execuţiei interogării:

```
SELECT NrFact, '2007-11-22', DataFact,
        '2007-11-22' - DataFact AS Zile_Scurse1
FROM facturi
```

NRFACT	2	DATAFACT	ZILE_SCURSE1
1.111	2007-11-22	01.08.2007	321
1.112	2007-11-22	01.08.2007	321
1.113	2007-11-22	01.08.2007	321
1.114	2007-11-22	01.08.2007	321
1.115	2007-11-22	02.08.2007	320
1.116	2007-11-22	02.08.2007	320
1.117	2007-11-22	03.08.2007	319
1.118	2007-11-22	04.08.2007	318
1.119	2007-11-22	07.08.2007	315
1.120	2007-11-22	07.08.2007	315
1.121	2007-11-22	07.08.2007	315
1.122	2007-11-22	07.08.2007	315
2.111	2007-11-22	14.08.2007	308
2.112	2007-11-22	14.08.2007	308
2.113	2007-11-22	14.08.2007	308
2.115	2007-11-22	15.08.2007	307
2.116	2007-11-22	15.08.2007	307
2.117	2007-11-22	16.08.2007	306
2.118	2007-11-22	16.08.2007	306
2.119	2007-11-22	21.08.2007	301

Figura 5.17. Diferenţa în ani, zile şi luni dintre două date calendaristice în DB2

coloana *Zile_Scurse1* (vezi figura 5.17) trebuie „citită" de la dreapta la stânga, ştiind că primele două cifre reprezintă numărul de zile, următoarele două numărul de luni, iar următoarele două, numărul de ani. Astfel, 321 de pe prima linie înseamnă 3 luni şi 21 de zile. Ştiind acum logica rezultatului, afişarea sub altă formă poate fi realizată prin funcţii de extragere a unui şir de caractere.

Despre alte funcţii dedicate gestionării datelor calendaristice şi şirurilor de caractere vom discuta în capitolul următor.

5.4. Ordonări

Una dintre caracteristicile modelului relaţional este că nici ordinea atributelor, nici ordinea liniilor în relaţii nu influenţează conţinutului informaţional al unei tabele. În practică, însă, forma de prezentare a rezultatelor interogării este importantă. Spre exemplu, o listă a localităţilor este mult mai folositoare dacă se prezintă în ordine alfabetică. Ordonarea liniilor în rezultatul unei interogări este posibilă prin clauza ORDER BY.

Să se obţină lista localităţilor în ordine alfabetică.

SELECT DISTINCT Loc
FROM coduri_postale
ORDER BY Loc

Rezultatul se prezintă ca în figura 5.18.

LOC
Birlad
Focsani
Iasi
Pascani
Roman
Suceava
Timisoara
Vaslui

Figura 5.18. Localităţile ordonate alfabetic

Aranjarea se face implicit crescător (ASC). Prin opţiunea DESC, ordinea prezentării se inversează. În plus, se pot specifica mai multe coloane care să servească drept criterii suplimentare de ordonare. La valori egale ale primului atribut, intră în acţiune criteriul de „balotaj" care este al doilea atribut etc.

Să se obţină, în ordinea descrescătoare a indicativului judeţelor, lista localităţilor în ordinea crescătoare a denumirii.

SELECT Jud, Loc, CodPost
FROM coduri_postale
ORDER BY Jud DESC, Loc ASC

Rezultatul este cel din figura 5.19. Ultimul indicativ de judeţ (e vorba despre ordine alfabetică) din tabela CODURI_POŞTALE este VS (pentru Vaslui), deci primele localităţi afişate vor fi din acest judeţ; prin urmare, prima linie a rezultatului se referă la municipiul Bârlad, iar a doua la municipiul Vaslui.

JUD	LOC	CODPOST
VS	Birlad	706400
VS	Vaslui	706510
VS	Vaslui	706500
VN	Focsani	705310
VN	Focsani	705300
TM	Timisoara	701900
SV	Suceava	705800
NT	Roman	705550
IS	Iasi	700515
IS	Iasi	700510
IS	Iasi	700505
IS	Pascani	701150

Figura 5.19. Două criterii de ordonare

Să se afişeze facturile din luna septembrie 2007 în ordinea alfabetică a observaţiilor (atributul Obs):

```
SELECT *
FROM facturi
WHERE DataFact >= DATE'2007-09-01' AND DataFact <= DATE'2007-09-30'
ORDER BY Obs
```
[10]

După cum se observă în rezultatul (Oracle) din figura 5.20, la început sunt liniile cu valorile nenule ale Obs, apoi apar toate cele în care valoarea atributului de ordonare este NULL[11].

NRFACT	DATAFACT	CODCL	OBS
3116	10-09-2007	1007	Pretul propus initial a fost modificat
3112	01-09-2007	1005	Probleme cu transportul
3118	17-09-2007	1001	(null)
3117	10-09-2007	1001	(null)
3113	02-09-2007	1002	(null)
3111	01-09-2007	1001	(null)
3115	02-09-2007	1001	(null)

Figura 5.20. Ordonarea implicită pentru un atribut ce prezintă valori NULL

Standardul SQL prezintă două clauze, NULLS FIRST şi NULLS LAST prin care ne putem exprima preferinţa în materie de poziţionarea valorilor NULL la

10. În MS SQL Server şi DB2 se elimină cele două cuvinte DATE.
11. În MS SQL Server, primele linii sunt cele în care valorile atributului Obs sunt NULL, în timp ce DB2 ordonează liniile ca în Oracle.

ordonare. Astfel, dacă în rezultat, preferăm să se înceapă cu valorile NULL, modificăm uşor interogarea de mai sus:

```
SELECT *
FROM facturi
WHERE datafact >= DATE'2007-09-01' AND datafact <= DATE'2007-09-30'
ORDER BY obs NULLS FIRST
```

Cum era de aşteptat, ordinea este acum cea din figura 5.21. Interogarea funcţionează în Oracle, dar nu şi în PostgreSQL înainte de versiunea 8.3 şi nici în SQL Server sau DB2 (toate versiunile), unde avem nevoie de o funcţie de conversie a valorilor nule (COALESCE) sau o structura alternativă de tip CASE. Începând cu versiunea 8.3, în PostgreSQL, lucrurile s-au remediat, interogarea de mai sus fiind funcţională.

O problemă (probabil) secundară ţine de modul în care specificăm un criteriu de ordonare, în condiţiile în care criteriul nu e un atribut, ci o expresie (coloană calculată). Ne interesează liniile facturii 1111, ordonate descrescător după valoarea fără TVA. Criteriul de ordonare se poate scrie în DB2, Oracle, PostgreSQL, SQL Server în mai multe moduri:

NRFACT	DATAFACT	CODCL	OBS
3111	01-09-2007	1001	(null)
3115	02-09-2007	1001	(null)
3117	10-09-2007	1001	(null)
3113	02-09-2007	1002	(null)
3118	17-09-2007	1001	(null)
3116	10-09-2007	1007	Pretul propus initial a fost modificat
3112	01-09-2007	1005	Probleme cu transportul

Figura 5.21. Modificarea ordonării pentru un atribut ce prezintă valori NULL

- Titulatura coloanei calculate, stabilită prin clauza AS:

```
SELECT liniifact.*, Cantitate * PretUnit AS ValLinie_FaraTVA
FROM liniifact
WHERE NrFact=1111
ORDER BY ValLinie_FaraTVA DESC
```

- Expresia propriu-zisă:

```
SELECT liniifact.*, Cantitate * PretUnit AS ValLinie_FaraTVA
FROM liniifact
WHERE NrFact=1111
ORDER BY Cantitate * PretUnit DESC
```

- Numărul de ordine al coloanei: ştiind că LINIIFACR are cinci atribute, expresia ar fi a şasea coloană:

```
SELECT liniifact.*, Cantitate * PretUnit AS ValLinie_FaraTVA
FROM liniifact
WHERE NrFact=1111
ORDER BY 6 DESC
```

5.5. Operatorii BETWEEN, OVERLAPS şi IN

Pentru formularea predicatului de selecţie, SQL permite utilizarea, pe lângă „clasicii" >, ≥, <, ≤, =, ≠, şi a altor operatori, dintre care ne vom opri la: BETWEEN (între, cuprins între), OVERLAPS (se suprapune) şi IN (în), la care se vor adăuga LIKE şi SIMILAR, discutaţi în paragraful 5.6, şi IS NULL prezentat în paragraful 8.2.

Operatorul BETWEEN este util pentru definirea intervalelor de valori. Ne reîntoarcem în capitolul 2, exemplul 3 (*Care sunt facturile emise în perioada 2-5 august 2007?*). Pe lângă soluţia formulată anterior în acest capitol, vom prezenta noi variante PostgreSQL şi Oracle.

- Soluţie „standard" - literalii de tip dată calendaristică sunt specificaţi prin clauza DATE:

```
SELECT *
FROM facturi
WHERE DataFact BETWEEN DATE'2007-08-02' AND DATE'2007-08-05'
```

- Soluţie DB2/PostgreSQL/SQL Server - literalii de tip dată se pot scrie direct:

```
SELECT * FROM facturi
WHERE DataFact BETWEEN '2007-08-02' AND '2007-08-05'
```

- Soluţia Oracle/PostgreSQL - specificare prin funcţia TO_DATE:

```
SELECT * FROM facturi
WHERE DataFact BETWEEN TO_DATE('02/08/2007','DD/MM/YYYY')
        AND TO_DATE('05/08/2007','DD/MM/YYYY')
```

Intervalul precizat începe obligatoriu cu limita minimă. Dacă am schimba interogarea de mai sus inversând cele două valori, varianta:

```
SELECT * FROM facturi
WHERE DataFact BETWEEN DATE'2007-08-05' AND DATE'2007-08-02'
```

nu returnează nimic.

BETWEEN poate fi folosit şi pentru atribute de tip şir de caractere. *Să se obţină, în ordinea judeţelor, lista localităţilor cu indicativul judeţului cuprins între IS (Iaşi) şi TM (Timiş).*

```
SELECT Jud, Loc, CodPost
FROM coduri_postale
WHERE Jud BETWEEN 'IS' AND 'TM'
ORDER BY Jud, Loc
```

Se poate folosi şi negaţia operatorului. O consecinţă interesantă este că, atunci când intervalul calendaristic este precizat defectuos, interogarea extrage întreg conţinutul tabelei, ca în acest exemplu:

```
SELECT *
FROM facturi
WHERE DataFact NOT BETWEEN DATE'2007-08-05' AND DATE'2007-08-02'
```

Pentru lucrul cu intervale calendaristice/temporale, SQL-ul prezintă un operator special - OVERLAPS. Acesta verifică dacă un interval calendaristic se suprapune (intersectează), fie şi parţial, cu alt interval ce serveşte drept argument:

SELECT *

FROM facturi

WHERE (DataFact, DataFact + INTERVAL '20 DAYS') OVERLAPS

(DATE'2007-09-15', DATE'2007-10-02')

Interogarea extrage facturile pentru care intervalul format din ziua de facturare (DataFact) şi următoarele 20 de zile se „intersectează", măcar pentru o zi, cu intervalul 15 septembrie - 2 octombrie 2007. Rezultatul este cel din figura 5.22. Operatorul OVERLAPS este implementat, deocamdată, numai în PostgreSQL.

nrfact numeric(8,0)	datafact date	codcl numeric(6,0)	obs character varying(50)
3111	2007-09-01	1001	
3112	2007-09-01	1005	Probleme cu transportul
3113	2007-09-02	1002	
3115	2007-09-02	1001	
3116	2007-09-10	1007	Pretul propus initial a fost modificat
3117	2007-09-10	1001	
3118	2007-09-17	1001	

Figura 5.22. Facturile pentru care intervalul calendaristic dintre data emiterii şi următoarele 20 de zile se suprapune cu intervalul 15 septembrie – 2 octombrie 2007

Operatorul IN se recomandă în predicatele ce verifică dacă valoarea unui atribut este încadrabilă într-o listă de valori dată. În locul folosirii în exces a operatorului OR, este mai elegant să se apeleze la serviciile operatorului IN. Forma generală: **expresie1 IN (expresie2, expresie3, ...)**.

Rezultatul evaluării unui predicat ce conţine acest operator va fi *adevărat* dacă valoarea expresiei1 este egală cu (cel puţin) una din valorile: expresie2, expresie3.

Care sunt localităţile din judeţele Iaşi (IS), Vaslui (VS) şi Timiş (TM)?

- Fără operatorul IN:

  SELECT DISTINCT Loc, Jud

  FROM coduri_postale

  WHERE Jud = 'IS' OR Jud = 'VS' OR Jud = 'TM'

  ORDER BY Jud, Loc

- Cu operatorul IN:

  SELECT DISTINCT Loc, Jud

  FROM coduri_postale

  WHERE Jud IN ('IS', 'VS', 'TM')

  ORDER BY Jud, Loc

Care sunt facturile întocmite pe 1, 3 şi 7 august 2007?

SELECT *

FROM facturi

WHERE DataFact IN (DATE'2007-08-01', DATE'2007-08-03', DATE'2007-08-07')

Despre câteva probleme legate de valorile NULL o să avem ocazia să discutăm în capitolul 8. Deocamdată, să introducem o valoare NULL în lista-argument a operatorului. Rezultatele interogării:

SELECT *

FROM facturi

WHERE DataFact IN (DATE'2007-08-01', DATE'2007-08-03',

DATE'2007-08-07', NULL)

nu diferă cu nimic de cel al SELECT-ului fără NULL: (SELECT * FROM facturi WHERE DataFact IN (DATE'2007-08-01', DATE'2007-08-03', DATE'2007-08-07')). În schimb, dacă folosim NOT IN în combinaţie cu un NULL în listă, lucrurile stau oarecum diferit. Fraza:

SELECT *

FROM facturi

WHERE DataFact NOT IN (DATE'2007-08-01',DATE'2007-08-03',DATE'2007-08-07')

extrage toate facturile din *alte zile decât* 1, 3 şi 7 august 2007, în timp ce rezultatul interogării:

SELECT *

FROM facturi

WHERE DataFact NOT IN (DATE'2007-08-01', DATE'2007-08-03',

DATE'2007-08-07', NULL)

nu conţine nicio linie.

Includerea valorilor NULL în listă este controlată strict când construim şi lansăm interogările direct, aşa cum procedăm în această lucrare. Există însă aplicaţii în care lista de valori este alcătuită din variabile (preluate din formulare), iar în anumite circumstanţe, datorită datelor introduse de utilizator, este posibil ca cel puţin una dintre variabile să se afle în situaţia de nulitate.

DB2-ul este reticent la introducerea valorilor NULL într-o listă-argument a operatorului IN, afişând un mesaj de eroare ca în figura 5.23.

```
------------------------------ Commands Entered ------------------------------
SELECT *
FROM facturi
WHERE DataFact IN ('2007-08-01', '2007-08-03', '2007-08-07', NULL);
------------------------------------------------------------------------------
SELECT * FROM facturi WHERE DataFact IN ('2007-08-01', '2007-08-03', '2007-08-07', NULL)
SQL0206N  "NULL" is not valid in the context where it is used.  SQLSTATE=42703
```

Figura 5.23. Opoziţia DB-ului (neconstructivă) la inserarea unui NULL într-o listă

5.6. Comparații inexacte. LIKE și SIMILAR

Deseori suntem în postura, oarecum ingrată, de a nu şti cu exactitate cum se numeşte un client sau un produs ori, pur şi simplu, unei persoane îi cunoaştem numai unul dintre prenume etc. Sunt situații pentru rezolvarea cărora a fost gândit operatorul LIKE. Operatorul LIKE permite compararea unui atribut (expresii) cu un literal utilizând o „mască" (şablon) construită cu ajutorul specificatorilor multipli % şi _. Simbolurile *procent* şi *liniuță_de_jos* (*underscore*, diferită de cratimă sau *liniuța de unire*) sunt denumite şi jokeri. Procentul substituie un şir de lungime variabilă, *0-n* caractere, în timp ce liniuța un singur caracter.

Care dintre firmele-client sunt societăți cu răspundere limitată (SRL-uri)?

Întrucât în tabela CLIENȚI denumirea fiecărui client se termină cu forma sa de societate (SA, SRL etc.), se poate redacta consultarea:

SELECT *

FROM clienti

WHERE DenCl LIKE '%SRL'

Rezultatul obținut prin execuția interogării în (Microsoft) SQL Server Management Studio se prezintă ca în figura 5.24.

codcl	dencl	codfiscal	adresa	codpost	telefon
1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	NULL
1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	0235222222
1005	Client 5 SRL	R1005	NULL	701900	0256111111
1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	0256121212

Figura 5.24. Clienții SRL-uri

Ce clienți au numele (fără forma de societate şi spațiul dinainte) formate din opt caractere şi sunt societăți pe acțiuni?

SELECT *

FROM clienti

WHERE DenCl LIKE '_______ SA'

Cele şapte semne *underscore* (nu se observă, dar sunt şapte) plus spațiul care le urmează nu au efect vizibil/impresionant pentru baza noastră de date, deoarece în capitolul 3, leneş fiind, am denumit toți clienții într-o manieră simplistă - Client 1 SRL, Client 2 SA etc. Dacă am avea mai mulți clienți (sau măcar un „Client 10 SA"), atunci interogarea ar avea cu totul alt farmec.

Ce persoane au numele ce conține litera S pe a treia poziție?

SELECT *

FROM persoane

WHERE Nume LIKE '__s%'

Rezultatul este vizualizat în figura 5.25. Atenție, dacă există persoane al căror nume are litera S majusculă pe a treia poziție, acestea nu sunt extrase în rezultat!

CNP	NUME	PRENUME	ADRESA	SEX	CODPOST	TELACASA	TELBIROU	TELMOBIL	EMAIL
CNP2	Vasile	Ion	(null)	B	700505	234567	876543	094222223	Ion@a.ro
CNP6	Vasc	Simona	M.Eminescu, 13	F	701150	(null)	432109	094222227	(null)

Figura 5.25. Persoane cu litera „s" pe a treia poziţie a numelui

În asemenea situaţii, soluţia de mai jos este ceva mai sigură:

```
SELECT *
FROM persoane
WHERE Nume LIKE '_s%' OR Nume LIKE '_S%'
```

În numele căror persoane apare, măcar o dată, litera S (indiferent de poziţie/poziţii)?

De data aceasta, trebuie să folosim specificatorul % pe care îl plasăm şi înainte, şi după litera S. Rezultatul este cel din figura 5.26.

```
SELECT *
FROM persoane
WHERE Nume LIKE '%s%' OR Nume LIKE '%S%'
```

CNP	NUME	PRENUME	ADRESA	SEX	CODPOST	TELACASA	TELBIROU	TELMOBIL	EMAIL
CNP2	Vasile	Ion	(null)	B	700505	234567	876543	094222223	Ion@a.ro
CNP6	Vasc	Simona	M.Eminescu, 13	F	701150	(null)	432109	094222227	(null)
CNP8	Bogacs	Ildiko	I.V.Viteazu. 67	F	705550	890123	210987	094222229	(null)

Figura 5.26. Persoane al căror nume conţine litera „s (S)"

Care sunt persoanele ce trebuie felicitate de Sfântul Ioan?

Fireşte, am fi tentaţi să redactăm varianta:

```
SELECT *
FROM persoane
WHERE UPPER(Prenume) LIKE '%ION%'
```

Funcţia UPPER face conversia valorilor atributului Prenume în majuscule (vezi capitolul următor). Rezultatul este însă cel din figura 5.27.

cnp	nume	prenume	adresa	sex	codpost	telacasa	telbirou	telmobil	email
CNP2	Vasile	Ion	NULL	B	700505	234567	876543	094222223	Ion@a.ro
CNP4	Lazar	Caraion	M.Eminescu, 42	B	706500	456789	NULL	094222225	NULL
CNP5	Iurea	Simion	I.Creanga, 44 bis	B	706500	567890	543210	NULL	NULL

Figura 5.27. Tentativă ratată de a extrage „Ionii"

Interogarea nu şi-a atins ţinta, deoarece, pe de o parte, nu au fost extrase persoanele cu prenume ca Ioan, Ioana, Ioanid, iar, pe de altă parte, au fost eronat extrase şi persoanele cu prenumele Caraion şi Simion. Prin urmare, cele trei litere - ION trebuie să fie plasate la începutul prenumelui. La acest şablon se adaugă şi IOAN. Bun, dar dacă pe omul nostru îl cheamă Marius Ion (are două prenume)? Cele două şabloane trebuie să se găsească la începutul fiecărui cuvânt din atributul Prenume. Iar dacă avem în vedere că uneori cele două prenume se despart prin

cratimă (Marius-Ion), rezultă o mândreţe de interogare (răspunsul este prezentat în figura 5.28) care funcţionează în toate cele patru servere BD:

```
SELECT *
FROM persoane
WHERE UPPER(Prenume) LIKE 'ION%' OR UPPER(Prenume) LIKE 'IOAN%'
      OR UPPER(Prenume) LIKE '% ION%' OR
      UPPER(Prenume) LIKE '% IOAN%' OR
      UPPER(Prenume) LIKE '%-ION%' OR
      UPPER(Prenume) LIKE '%-IOAN%'
```

cnp	nume	prenume	adresa	sex	codpost	telacasa	telbirou	telmobil	email
CNP2	Vasile	Ion	NULL	B	700505	234567	876543	094222223	Ion@a.ro
CNP3	Popovici	Ioana	V.Micle, Bl.I, Sc.B,Ap.2	F	701150	345678	NULL	094222224	NULL
CNP7	Popa	Ioanid	I.Ion, Bl.H2, Sc.C, Ap.45	B	701900	789012	321098	NULL	NULL

Figura 5.28. Persoanele care trebuie felicitate de Sfântul Ioan

SQL Server poate folosi în şablonul clauzei LIKE, pe lângă % şi _, şi paranteze drepte, şi caracterul ^. Ne interesează toate persoanele ale căror prenume încep cu litera I sau V. În afara variantei standard:

```
SELECT * FROM persoane
WHERE UPPER(Prenume) LIKE 'I%' OR UPPER(Prenume) LIKE 'V%'
```

SQL Server acceptă şi varianta ceva mai scurtă:

```
SELECT *
FROM persoane
WHERE UPPER(Prenume) LIKE '[I,V]%'
```

Iar dacă ne-ar interesa toate persoanele, cu excepţia celor ale căror nume încep fie cu litera I, fie cu litera V, varianta generală ar fi:

```
SELECT *
FROM persoane
WHERE UPPER(Prenume) NOT LIKE 'I%' AND
UPPER(Prenume) NOT LIKE 'V%'
```

în timp ce SQL Server acceptă şi sintaxa:

```
SELECT * FROM persoane WHERE UPPER(Prenume) LIKE '[^I,V]%'
```

Tot aici putem nota operatorul ILIKE din PostgreSQL care are aceeaşi funcţionalitate ca LIKE, doar că nu se mai ţine cont dacă literele sunt majuscule sau minuscule şi, astfel, nu avem nevoie de funcţia UPPER. Interogarea „Ionilor" se poate rescrie ceva mai simplu:

```
SELECT *
FROM persoane
WHERE Prenume ILIKE 'ION%' OR Prenume ILIKE 'IOAN%' OR
      Prenume ILIKE '% ION%' OR Prenume ILIKE '% IOAN%'
      OR Prenume ILIKE '%-ION%' OR Prenume ILIKE '%-IOAN%'
```

Deşi rare, există cazuri când printre caracterele căutate în valorile unui atribut şir de caractere se găseşte chiar unul dintre cele două şabloane, _ sau %. Dacă, spre exemplu, ne interesează toţi clienţii care conţin simbolul % în numele lor (s-ar putea ca, la un moment dat, să avem în bază clienţi de genul „Procentul vesel % SRL"). Soluţia este:

```
SELECT *
FROM clienti
WHERE DenCl LIKE '%!%%' ESCAPE '!'
```

Datorită primului şi ultimului simbol procent, poziţia caracterului căutat (în cazul nostru, chiar %) nu prezintă importanţă: poate fi prima, ultima sau oricare între prima şi ultima. Rezultatul corect este obţinut graţie unui caracter declarat prin clauza ESCAPE. Acesta este, în interogarea noastră, semnul mirării (!), dar putea fi oricare altul. Prin clauza ESCAPE s-a indicat SQL-ului că procentul ce urmează simbolului „!" nu este joker, ci are regim de caracter oarecare, ce trebuie căutat în tabelă ca atare.

Standardul SQL:1999 a introdus un mecanism mai elegant de căutare preluat din sistemul de operare UNIX (Posix), mecanism bazat pe operatorul SIMILAR şi implementat doar în PostgreSQL. Cele două caractere folosite în construirea „măştilor" sunt tot „%" şi „_". Modul de construire a expresiei de căutare este însă mult mai generos, principalele caractere disponibile fiind:

| - pentru structuri alterative;

* - pentru repetarea şirului (de caractere) precedent de zero sau mai multe ori;

+ - pentru repetarea şirului precedent o dată sau de mai multe ori;

() - pentru gruparea mai multor şiruri de caractere într-o unitate logică;

[] - pentru indicarea unei clase de caractere.

Să începem cu câteva chestiuni simple. Aflarea *firmelor-client care sunt societăţi cu răspundere limitată* se poate realiza în modul prezentat la începutul paragrafului, înlocuind LIKE cu SIMILAR TO:

```
SELECT *
FROM clienti
WHERE DenCl SIMILAR TO '%SRL'
```

Expresia-şablon poate folosi paranteze:

```
SELECT * FROM clienti WHERE DenCl SIMILAR TO '(%SRL)'
```

sau

```
SELECT * FROM clienti WHERE DenCl SIMILAR TO '%(SRL)'
```

Prin folosirea caracterului „ | " (bară verticală) evităm folosirea operatorului OR. Reluând ultima soluţie care ar extrage „candidaţii" la felicitări de Sfântul Ioan, prin folosirea noului caracter, interogarea se simplifică vizibil, cel puţin ca lungime:

```
SELECT *
FROM persoane
WHERE UPPER(Prenume) SIMILAR TO
        'ION% | IOAN% | % ION% | % IOAN | %-ION% | %-IOAN%'
```

Deşi nu prea are rost, putem folosi şi parantezele:

```
SELECT * FROM persoane
WHERE UPPER(Prenume) SIMILAR TO
'(ION%) | (IOAN%) | (% ION%) | (% IOAN) | (%-ION%) | (%-IOAN%)'
```

Continuăm cu o problemă ceva mai exotică. Vrem să aflăm *clienţii care au sediul la un număr care conţine numai cifrele 2, 3 sau 5*. Ştiind că numărul urmează unui spaţiu de după numele străzii, construim şablonul după cum urmează:

```
SELECT *
FROM clienti
WHERE Adresa SIMILAR TO '% [235]+'
```

Ne interesează doar cifrele 2, 3 şi 5, deci excludem combinaţiile acestora cu alte cifre. Lucrul acesta se indică prin plasarea lor între paranteze drepte. Semnul plus din finalul şablonului indică repetarea de oricâte ori (de 1-n ori) sau, altfel spus, se vor selecta numai liniile în care cifra 2, 3 sau 5 apare măcar o dată în combinaţie cu ea însăşi sau celelate două. Rezultatul interogării PostgreSQL conţine trei linii, ca în figura 5.29.

codd numeric(6,0)	dencl character vai	codfiscal character(9)	adresa character varying(40)	codpost character(6)	telefon character va
1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	035-222222
1006	Client 6 SA	R1006	Pacientei, 33	705550	
1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	056-121212

Figura 5.29. Clienţii ce au numărul de la adresa alcătuit numai din una sau mai multe dintre cifrele 2, 3 sau 5

Dacă nu am fi folosit în finalul şablonului semnul +, adică:

```
SELECT *
FROM clienti
WHERE Adresa SIMILAR TO '% [235]'
```

rezultatul ar fi conţinut doar un rând (vezi figura 5.30), întrucât doar Clientul 7 are numărul de la adresă alcătuit dintr-o singură cifră (2).

codd numeric(6,0)	dencl character vai	codfiscal character(9)	adresa character varying(40)	codpost character(6)	telefon character va
1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	056-121212

Figura 5.30. Clienţii ce au numărul de la adresa alcătuit numai din una dintre cifrele 2, 3 sau 5

Semnul „+" indică repetarea de *1-n* ori a cifrei/cifrelor din şablon, în timp ce asteriscul („*") indică repetarea de *0-n* ori. Posix („sortimentul" de Unix din care provine mecanismul SIMILAR TO) are opţiuni pentru a indica numărul exact de repetări ale caracterului specificat, însă nici standardele SQL şi nici PostgreSQL-ul nu au încă această facilitate.

O problemă a soluţiilor de mai sus iese la iveală dacă ne propunem să extragem clienţii cu adrese la numere alcătuite numai din cifrele 1 şi 3. Interogarea:

```
SELECT *
FROM clienti
WHERE Adresa SIMILAR TO '(% [13]+)'
```

extrage doar clientul 6 (vezi figura 5.31), deşi clientul 1 are sediul pe strada Tranziţiei, nr. 13 bis.

codcl numeric(6,0)	dencl character var	codfiscal character(9)	adresa character varying(40)	codpost character(6)	telefon character va
1006	Client 6 SA	R1006	Pacientei, 33	705550	

Figura 5.31. Un singur client cu numărul de la adresă alcătuit numai din una sau din ambele cifre 1 şi 3 (deşi sunt doi)

Necazul decurge din faptul că numărul 13 se continuă cu „bis", aşa că avem nevoie de modificarea şablonului:

```
SELECT *
FROM clienti
WHERE Adresa SIMILAR TO '(% [13]+) | (% [13]+ bis)'
```

De data aceasta, rezultatul este corect, după cum se poate observa în figura 5.32. Am folosit secvenţa alternativă (operatorul „ | ").

codcl numeric(6,0)	dencl character var	codfiscal character(9)	adresa character varying(40)	codpost character(6)	telefon character var
1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	
1006	Client 6 SA	R1006	Pacientei, 33	705550	

Figura 5.32. Cei doi clienţi cu numărul de la adresă alcătuit numai din una sau din ambele cifre 1 şi 3

Care sunt persoanele care stau la blocuri ale căror nume este alcătuit dintr-o literă şi o cifră (de exemplu, H2, B5, A9)?

Pentru litera din denumirea blocului se poate folosi jokerul *underscore*. În schimb, substituirea unei cifre se poate face incluzând între paranteze drepte intervalul 0-9, ca în interogarea următoare al cărei rezultat este afişat în figura 5.33.

```
SELECT *
FROM persoane
WHERE Adresa SIMILAR TO '%Bl._[0-9]%'
```

Din păcate, celelalte trei servere nu au implementat SIMILAR TO, ci doar câteva funcţii cu acţiune similară.

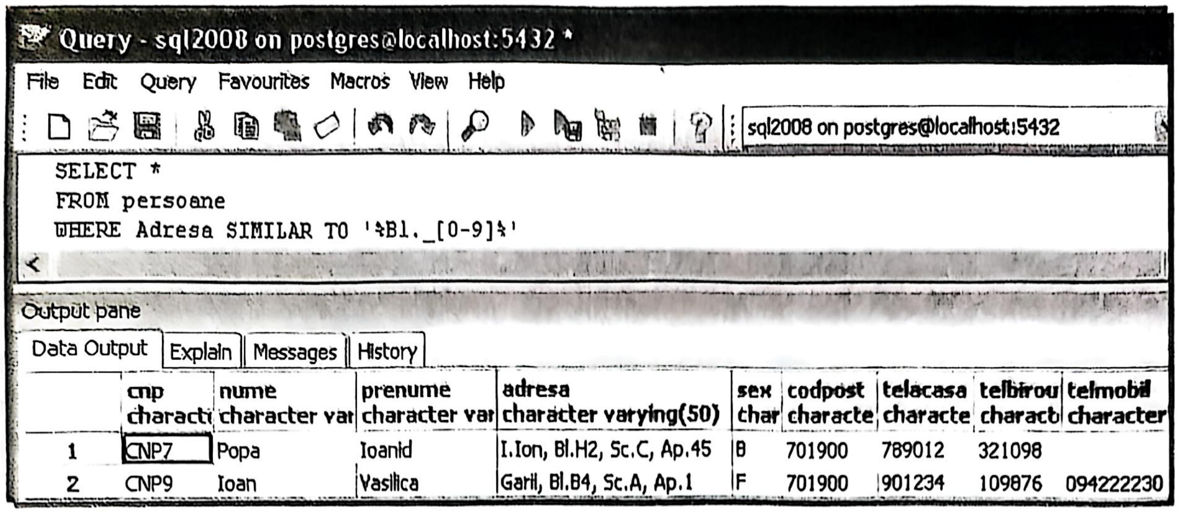


Figura 5.33. Predicat SIMILAR TO pentru extragerea în PostgreSQL a domiciliaților în blocuri ale căror nume cuprinde o literă şi o cifră

5.7. Joncțiuni interne

SQL prezintă clauze sau operatori speciali pentru aproape toate tipurile de joncțiuni discutate în capitolul 4. O joncțiune internă este o combinație de produs cartezian şi selecție. În consecință, pentru theta-joncționarea relațiilor R1 şi R2 din *exemplul 10* al algebrei relaționale (paragraful 4.4.4, figura 4.17) se poate scrie:

```
SELECT *
FROM r1, r2
WHERE r1.A >= r2.E
```

iar pentru echijoncționarea din *exemplul 11* (figura 4.18):

```
SELECT *
FROM r1, r2
WHERE r1.A = r2.E
```

În toate cele patru servere este implementată opțiunea propusă de SQL-92, INNER JOIN (joncțiune internă). Astfel, cele două soluții de mai sus pot fi rescrise după cum urmează:

```
SELECT *
FROM r1 INNER JOIN r2 ON r1.A >= r2.E
```

respectiv:

```
SELECT *
FROM r1 INNER JOIN r2 ON r1.A >= r2.E
```

Reluăm, pentru comparație, câteva exemple din algebra relațională.

Exemplul 13. Să se obțină, pentru fiecare localitate, codurile poştale, denumirea, indicativul şi denumirea județului şi regiunea din care face parte – paragraful 4.4.4.

Varianta 1 (SQL-89):

```
SELECT CodPost, Loc, coduri_postale.Jud, Judet, Regiune
```

FROM coduri_postale, judete

WHERE coduri_postale.Jud = judete.Jud

Varianta 2 (SQL-92):

SELECT CodPost, Loc, coduri_postale.Jud, Judet, Regiune

FROM coduri_postale INNER JOIN judete

ON coduri_postale.Jud = judete.Jud

Numai atributul Jud a fost prefixat de numele tabelei din care provine. Operațiunea este obligatorie atunci când câmpul există în două sau mai multe dintre tabelele enumerate în clauza FROM. Astfel, dacă nu prefixăm atributul Jud în clauza SELECT vom obține un mesaj de genul *Coloană definită ambiguu/Column ambigously defined* (în funcție de limba specificată la instalare) - vezi figura 5.34. O bilă albă pentru Oracle SQL Developer care afişează mesajul şi în română şi în engleză.

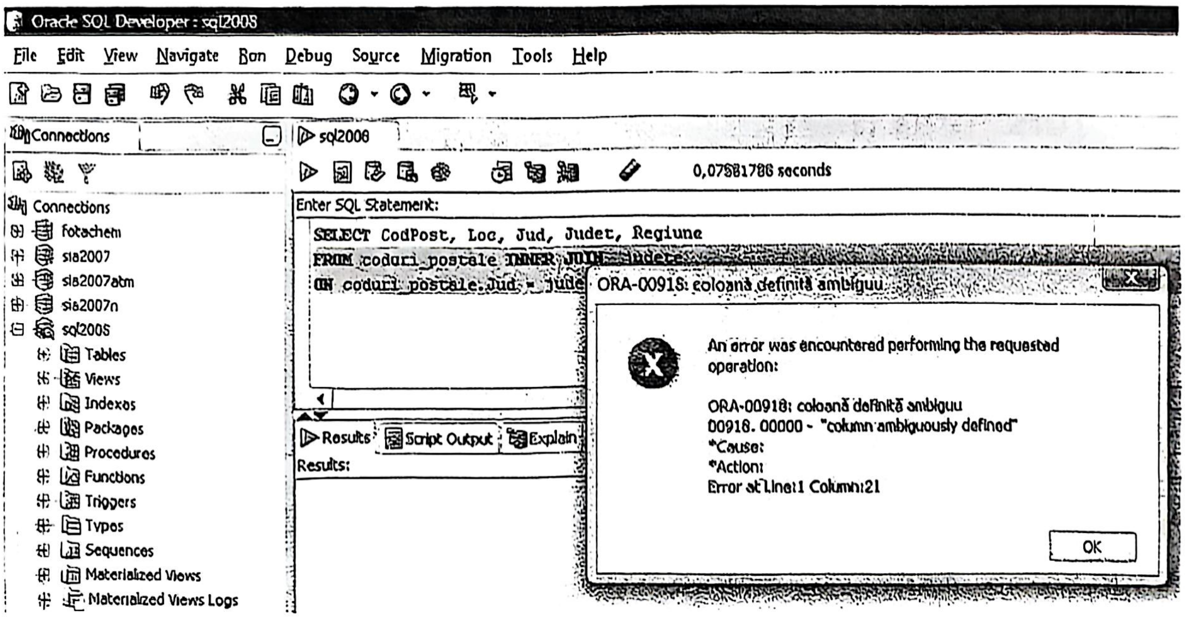


Figura 5.34. Eroare (de ambiguitate) la nespecificarea tabelei pentru un atribut ce apare în două sau mai multe relații din clauza INNER JOIN

Exemplul 14. Care sunt codurile poştale ale localităților din Banat? - paragraful 4.4.4 - figura 4.20.

Varianta SQL-1:

SELECT CodPost, Loc, coduri_postale.Jud, Judet, Regiune

FROM coduri_postale, judete

WHERE coduri_postale.Jud = judete.Jud AND Regiune= 'Banat'

În clauza WHERE, predicatului de selecție pentru realizarea joncțiunii i s-a adăugat secvența de test a regiunii.

Varianta SQL-2:

SELECT CodPost, Loc, coduri_postale.Jud, Judet, Regiune

FROM coduri_postale INNER JOIN judete ON coduri_postale.Jud = judete.Jud

WHERE Regiune='Banat'

Avantajul celei de-a doua variante ține de separarea condiției ce ține de regiune de condiția legată de joncțiunea propriu-zisă.

Exemplul 15. În ce zile s-a vândut produsul cu denumirea „Produs 1"? - paragraful 4.4.4.

```
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
WHERE DenPr = 'Produs 1'
```

De notat folosirea clauzei DISTINCT pentru eliminarea eventualelor dubluri.

Exemplul 16. În ce județe s-a vândut produsul cu denumirea „Produs 1" în perioada 3-5 august 2007? - paragraful 4.4.4.

```
SELECT DISTINCT Judet
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
        INNER JOIN clienti ON facturi.CodCl = clienti.CodCl
        INNER JOIN coduri_postale
        ON clienti.CodPost=coduri_postale.CodPost
        INNER JOIN judete ON coduri_postale.Jud = judete.Jud
WHERE DenPr = 'Produs 1' AND DataFact BETWEEN
        DATE'2007-08-03' AND DATE'2007-08-05'
```

Şi în acest exemplu este recomandată folosirea clauzei DISTINCT[12].

Exemplul 17. În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"? - paragraful 4.4.4.

```
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
WHERE DenPr = 'Produs 1'
        INTERSECT
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
WHERE DenPr = 'Produs 2'
```

12. Nu uitați să eliminați cele două cuvintele DATE pentru ca interogarea să funcționeze în DB2 şi SQL Server.

Interesant este că putem include predicatul de selecţie chiar în clauza de joncţiune:

```
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact
        ON produse.CodPr = liniifact.CodPr AND DenPr = 'Produs 1'
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
INTERSECT
SELECT DISTINCT DataFact
FROM produse INNER JOIN liniifact
        ON produse.CodPr = liniifact.CodPr AND DenPr = 'Produs 2'
INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
```

Ar mai fi o soluţie bazată pe joncţiunea tabelei cu ea însăşi, lucru pe care îl vom discuta în curând în acest paragraf.

Facem ce facem şi ne legăm de lucruri minore. În cazul în care joncţionăm două tabele, iar în clauza SELECT folosim asteriscul („*") pentru a semnaliza includerea tuturor atributelor în rezultat, ce nume vor primi atributele care sunt prezente în două sau mau multe dintre tabelele din clauza FROM? Regula este... că nu e nicio regulă. Execuţia interogării:

```
SELECT *
FROM judete INNER JOIN coduri_postale ON judete.Jud=coduri_postale.Jud
```

în Oracle SQL Developer duce la o titulatură a coloanelor ca în figura 5.35. Fiind grijuliu în ceea ce priveşte unicitatea denumirii fiecărui atribut, SQL Developer atribuie „celui de-al doilea Jud" titulatura JUD_1, iar DB2-ul titulatura JUD1.

JUD	JUDET	REGIUNE	CODPOST	LOC	JUD_1
IS	Iasi	Moldova	700505	Iasi	IS
IS	Iasi	Moldova	700510	Iasi	IS
IS	Iasi	Moldova	701150	Pascani	IS
IS	Iasi	Moldova	700515	Iasi	IS
NT	Neamt	Moldova	705550	Roman	NT
SV	Suceava	Moldova	705800	Suceava	SV

Figura 5.35. Selectarea tuturor atributelor din două tabele joncţionate în Oracle SQL Developer

În PostgreSQL pgAdmin, nu e nicio precupare pentru unicitatea denumirii coloanelor în rezultat, aşa încât toate coloanele care se repetă au acelaşi nume - vezi figura 5.36. Similar stau lucrurile şi în SQL Server Management Studio. Am uitat să vă spun că în DB2 şi SQL Server în loc de INNER JOIN se poate folosi, simplu, JOIN.

Pentru moment, problema denumirii coloanelor este una fără semnificaţii majore, însă la definirea ad-hoc, în clauza FROM a unei tabele pe baza unei sub-consultări, denumirile coloanelor trebuie gestionate cu mai mare grijă.

jud character(2)	judet character var	regiune character var	codpost character(6)	loc character var	jud character(2)
IS	Iasi	Moldova	700505	Iasi	IS
IS	Iasi	Moldova	700510	Iasi	IS
IS	Iasi	Moldova	700515	Iasi	IS
IS	Iasi	Moldova	701150	Pascani	IS
VS	Vaslui	Moldova	706500	Vaslui	VS

Figura 5.36. Selectarea tuturor atributelor din două tabele joncționate în PostgreSQL pgAdmin

În prima ediție a cărții am evitat cu grijă atât exemple de theta-joncțiune, cât și clauza NATURAL JOIN dedicată, cum era de așteptat, joncțiunii interne naturale. În privință joncțiunii naturale trebuie restabilit adevărul istoric, și anume că în SQL există operatorul NATURAL JOIN pe care l-am descris în paragraful 4.4.4 (*exemplul 12*, figura 4.19). Avantajul acestui operator ține atât de lungimea interogării (automat joncționarea se realizează după toate atributele comune tabelelor din clauza FROM), cât și de păstrarea în rezultat doar a unei singure apariții pentru fiecare atribut comun (de joncționare). Revenim la interogarea din figura 5.35. Folosind operatorul NATURAL JOIN, implementat în DB2, Oracle, PostgreSQL (nu, însă, și în MS SQL Server):

```
SELECT *
FROM judete NATURAL JOIN coduri_postale
```

obținem un rezultat ca în figura 5.37.

JUD	JUDET	REGIUNE	CODPOST	LOC
IS	Iasi	Moldova	700505	Iasi
IS	Iasi	Moldova	700510	Iasi
IS	Iasi	Moldova	701150	Pascani
IS	Iasi	Moldova	700515	Iasi
NT	Neamt	Moldova	705550	Roman
SV	Suceava	Moldova	705800	Suceava
TM	Timis	Banat	701900	Timisoara
VN	Vrancea	Moldova	705310	Focsani
VN	Vrancea	Moldova	705300	Focsani
VS	Vaslui	Moldova	706400	Birlad
VS	Vaslui	Moldova	706500	Vaslui
VS	Vaslui	Moldova	706510	Vaslui

Figura 5.37. O joncțiune internă (care funcționează) în Oracle

Exemplul 17. Ce clienți au cumpărat și „Produs 2", și „Produs 3", dar nu au cumpărat „Produs 5"? - paragraful 4.4.4.

Cu noul operator pentru joncțiune naturală, efortul de scriere a interogării se diminuează în oarecare măsură:

```
SELECT DISTINCT DenCl
FROM produse NATURAL JOIN liniifact NATURAL JOIN facturi
        NATURAL JOIN clienti
```

```
WHERE DenPr = 'Produs 2'
INTERSECT
SELECT DISTINCT DenCl
FROM produse NATURAL JOIN liniifact NATURAL JOIN facturi
        NATURAL JOIN clienti
WHERE DenPr = 'Produs 3'
EXCEPT
SELECT DISTINCT DenCl
FROM produse NATURAL JOIN liniifact NATURAL JOIN facturi
        NATURAL JOIN clienti
WHERE DenPr = 'Produs 5'
```

Dacă e să dăm crezare figurii 5.38 interogarea merge fără cusur. Aşa stau lucrurile în PostgreSQL.

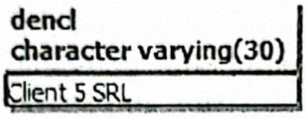


Figura 5.38. O joncţiune internă (care funcţionează) în Oracle

Executată în Oracle SQL Developer şi schimbând EXCEPT cu MINUS, aceeaşi interogare nu produce niciun rezultat – vezi figura 5.39.

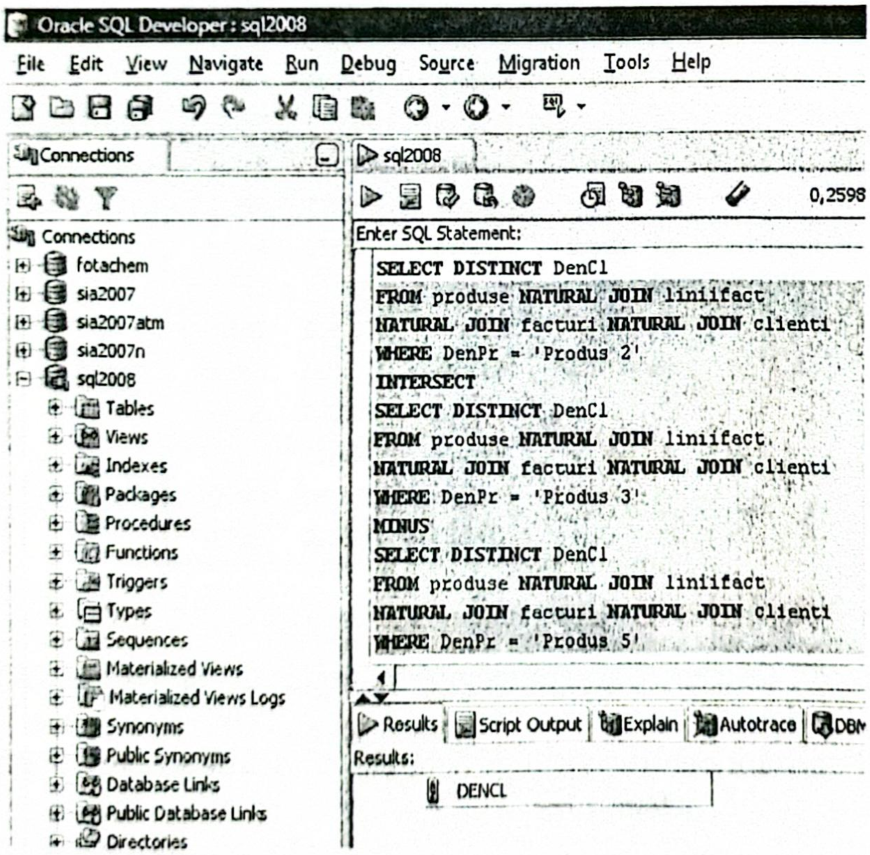


Figura 5.39. O joncţiune internă care nu funcţionează în Oracle

Gândindu-mă la celebrul X-Files sau alte ciudățenii, am încercat alte variante de joncțiune naturală în Oracle. Dintre rezultatele tulburătoare obținute, vă supun atenției două interogări plasate față în față care ar trebui, teoretic, să obțină același număr de linii:

SELECT *	SELECT dencl
FROM produse	FROM produse
NATURAL JOIN liniifact	NATURAL JOIN liniifact
NATURAL JOIN facturi	NATURAL JOIN facturi
NATURAL JOIN clienti	NATURAL JOIN clienti
WHERE DenPr = 'Produs 1' AND	WHERE DenPr = 'Produs 1' AND
datafact BETWEEN DATE'2007-08-01'	datafact BETWEEN DATE'2007-08-01'
AND DATE'2007-08-03'	AND DATE'2007-08-03'

Ei bine, nici vorbă! Interogarea din stânga (SELECT *) extrage două linii în rezultat, în timp ce a doua (SELECT dencl) extrage nu mai puțin de 441 de linii! Nu am avut curiozitatea să văd ce funcție recursivă au putut aplica cei de la Oracle ca să obțină așa mândrețe de rezultat. Am inclus însă în figura 5.40 primele 8,5 coloane din rezultatul primei interogări (în stânga) și primele 4,5 linii din rezultatul celei de-a doua.

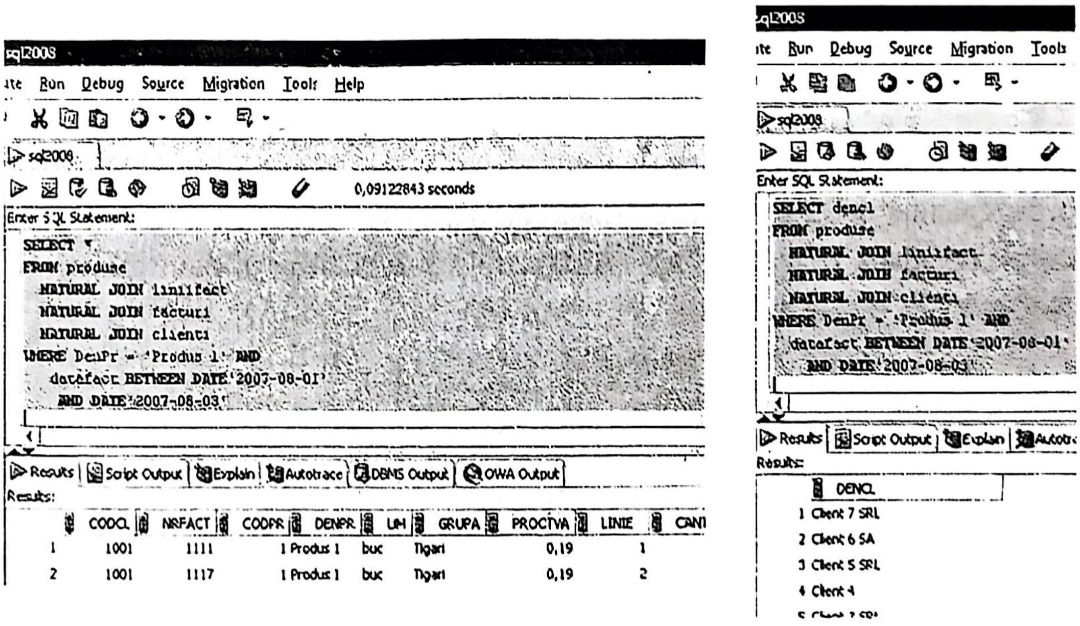


Figura 5.40. Surprize, suprize... în Oracle

Pentru exemplificarea theta-joncțiunii revenim la *exemplul 20* din finalul paragrafului 4.4.4. Echivalent soluției propuse în algebra relațională, se poate redacta următoarea interogare SQL:

```
SELECT dataora_examinare, numepacient, numedoctor
FROM triaj INNER JOIN garzi ON dataora_examinare
BETWEEN inceput_garda AND sfarsit_garda
        INNER JOIN doctori ON garzi.iddoctor = doctori.iddoctor
        INNER JOIN pacienti ON triaj.idpacient = pacienti.idpacient
ORDER BY 1
```

Rezultatul este cel din figura 5.41.

DATAORA_EXAMINARE	NUMEPACIENT	NUMEDOCTOR
Jan 3, 2008 7:18:00 AM 000000	Stroe Mihaela	Georgescu Mircea
Jan 3, 2008 8:45:00 AM 000000	Buzatu Corneliu	Georgescu Mircea
Jan 3, 2008 12:45:00 PM 000000	Spineanu Marius	Georgescu Mircea
Jan 3, 2008 8:45:00 PM 000000	Bagdasar Adela	Zahir Tudorel
Jan 4, 2008 1:28:00 AM 000000	Ionascu Ionelia	Zahir Tudorel
Jan 4, 2008 10:45:00 AM 000000	Spineanu Marius	Bostan Vasile
Jan 4, 2008 11:20:00 AM 000000	Stroescu Mihaela Oana	Bostan Vasile
Jan 4, 2008 10:45:00 PM 000000	Cazan Ana Maria	Popescu Ionela
Jan 5, 2008 6:18:00 AM 000000	Ionascu Ionelia	Georgescu Mircea

Figura 5.41. Rezultatul interogării în care este implicată theta-joncţiunea tabelelor TRIAJ şi GĂRZI

Dacă despre joncţiunile externe ne vom ocupa abia în capitolul 8, despre semijoncţiune lămurim lucrurile pe loc, dacă este, într-adevăr, ceva de lămurit. Pentru ilustrare, reluăm *exemplul* 22 din paragraful 4.4.6 - Care sunt codurile poştale (denumirea localităţii şi indicativul judeţului) în care există măcar un client?

Soluţia SQL este mai mult decât banală, întrucât presupune prefixarea asteriscului, în clauza SELECT, cu numele tabelei din care ne interesează valorile - CODURI_POŞTALE:

```
SELECT coduri_postale.*
FROM coduri_postale INNER JOIN clienti
        ON coduri_postale.CodPost = clienti.CodPost
```

În timp ce tabela CODURI_POŞTALE are 12 linii, rezultatul din figura 5.42 are doar 7.

CODPOST	LOC	JUD
700505	Iasi	IS
700505	Iasi	IS
701150	Pascani	IS
701900	Timisoara	TM
701900	Timisoara	TM
705550	Roman	NT
706500	Vaslui	VS

Figura 5.42. Semijoncţiunea dintre CODURI_POŞTALE şi CLIENŢI

5.8. Sinonime locale şi joncţiunea unei tabele cu ea însăşi

Lucrul cu nume lungi de tabele are marile avantaje al lejerităţii la „citire" şi al înţelegerii rapide a logicii de derulare a interogării. În schimb, destul de mulţi informaticieni nu agreează risipa de caractere (şi, implicit, de timp şi nervi) implicată de redactările „logoreice". Ambele părţi au dreptate în oarecare măsură (sesizaţi sindromul rabinului!). Astfel, în frazele SELECT, tabelelor li se pot asocia sinonime sau aliasuri mai scurte:

```
SELECT *
FROM produse P
        INNER JOIN liniifact LF ON P.CodPr = LF.CodPr
        INNER JOIN facturi F ON LF.NrFact = F.NrFact
```

Tabelei PRODUSE i s-a asociat sinonimul P, LINIIFACT LF, iar pentru FACTURI F. Sinonimele prefixează (când este necesar) numele atributelor în clauzele SELECT şi WHERE (eventual ORDER BY, GROUP BY).

Există situaţii în care utilizarea sinonimelor n-are nimic de-a face cu lenea/comoditatea sau depresiile. În afara interogărilor mai mult sau mai puţin corelate, pe care le vom discuta în capitolele viitoare, o operaţiune în care trebuie musai folosite sinonimele este joncţionarea tabelei cu ea însăşi sau, cum spunem noi, moldovenii, cu *dânsa-însăşi*[13].

Care sunt judeţele din regiunea ţării în care se află judeţul Iaşi?

O asemenea problemă se rezolvă dintr-o clipită folosind subconsultări. Noi, însă, ţinem morţiş să folosim doar ceea ce am discutat în acest capitol. Iată o soluţie interesantă:

```
SELECT j1.*
FROM judete j1
        INNER JOIN judete j2 ON j1.Regiune = j2.Regiune
WHERE j2.Judet = 'Iasi'
```

Să discutăm pe etape ceea ce se întâmplă. Joncţiunea a două instanţe ale tabelei JUDEŢE pe baza egalităţii valorilor atributului Regiune (vezi figura 5.43):

```
SELECT *
FROM judete j1
        INNER JOIN judete j2 ON j1.Regiune = j2.Regiune
```

va obţine un rezultat în care pe fiecare linie se află două judeţe (sau chiar acelaşi judeţ) care sunt din aceeaşi regiune[14].

---

13. Aceasta este una dintre glumele nesărate din prima ediţie.
14. Adevărul este că, pentru a obţine exact titulatura coloanelor din figură, interogarea are forma: SELECT j1.Jud AS "j1.Jud", j1.Judet AS "j1.Judet", j1.Regiune AS "j1.Regiune", j2.Jud AS "j2.Jud", j2.Judet AS "j2.Judet", j2.Regiune AS "j2.Regiune" FROM judete j1 INNER JOIN judete j2 ON j1.Regiune=j2.Regiune.

j1.Jud	j1.Judet	j1.Regiune	j2.Jud	j2.Judet	j2.Regiune
VS	Vaslui	Moldova	IS	Iasi	Moldova
SV	Suceava	Moldova	IS	Iasi	Moldova
NT	Neamt	Moldova	IS	Iasi	Moldova
VN	Vrancea	Moldova	IS	Iasi	Moldova
IS	Iasi	Moldova	IS	Iasi	Moldova
VS	Vaslui	Moldova	VN	Vrancea	Moldova
SV	Suceava	Moldova	VN	Vrancea	Moldova
NT	Neamt	Moldova	VN	Vrancea	Moldova
VN	Vrancea	Moldova	VN	Vrancea	Moldova
IS	Iasi	Moldova	VN	Vrancea	Moldova
VS	Vaslui	Moldova	NT	Neamt	Moldova
SV	Suceava	Moldova	NT	Neamt	Moldova
NT	Neamt	Moldova	NT	Neamt	Moldova
VN	Vrancea	Moldova	NT	Neamt	Moldova
IS	Iasi	Moldova	NT	Neamt	Moldova
VS	Vaslui	Moldova	SV	Suceava	Moldova
SV	Suceava	Moldova	SV	Suceava	Moldova
NT	Neamt	Moldova	SV	Suceava	Moldova
VN	Vrancea	Moldova	SV	Suceava	Moldova
IS	Iasi	Moldova	SV	Suceava	Moldova
VS	Vaslui	Moldova	VS	Vaslui	Moldova
SV	Suceava	Moldova	VS	Vaslui	Moldova
NT	Neamt	Moldova	VS	Vaslui	Moldova
VN	Vrancea	Moldova	VS	Vaslui	Moldova
IS	Iasi	Moldova	VS	Vaslui	Moldova
TM	Timis	Banat	TM	Timis	Banat

Figura 5.43. Autojoncţiunea tabelei JUDEŢE după atributul Regiune

Ştiind că pe o linie avem două judeţe (sau unul) din aceeaşi regiune, filtrăm una dintre instanţele tabelei JUDEŢE (a doua, în cazul nostru), păstrând numai liniile în care j2.Judet este Iaşi – vezi figura 5.44:

```
SELECT *
FROM judete j1 INNER JOIN judete j2 ON j1.Regiune=j2.Regiune
WHERE j2.Judet='Iasi'
```

j1.Jud	j1.Judet	j1.Regiune	j2.Jud	j2.Judet	j2.Regiune
IS	Iasi	Moldova	IS	Iasi	Moldova
VN	Vrancea	Moldova	IS	Iasi	Moldova
NT	Neamt	Moldova	IS	Iasi	Moldova
SV	Suceava	Moldova	IS	Iasi	Moldova
VS	Vaslui	Moldova	IS	Iasi	Moldova

Figura 5.44. Filtrarea liniilor din autojoncţiune

Acum vedem că cel de-al doilea „rând de coloane" - cel care corespunde instanței j2 - este de prisos, astfel încât prefixăm asteriscul cu aliasul (sinonimul local) j1, obținând înregistrările din figura 5.45.

JUD	JUDET	REGIUNE
IS	Iasi	Moldova
VN	Vrancea	Moldova
NT	Neamt	Moldova
SV	Suceava	Moldova
VS	Vaslui	Moldova

Figura 5.45. Rezultat final – „colegii" de regiune ai județului Iași

Acum, dacă tot ne-am chinuit așa de tare, revenim la exemplul 19 din algebra relațională: *ce facturi au fost emise în aceeași zi cu factura 1120?*

Pe calapodul soluției de mai sus, redactăm una care, pe deasupra, elimină din rezultat factura 1120 (vezi figura 5.46), deci răspunde cu exactitate enunțului:

```
SELECT F2.NrFact
FROM facturi F1 INNER JOIN facturi F2
        ON F1.DataFact = F2.DataFact AND F1.NrFact=1120
WHERE F2.NrFact <> 1120
```

Pentru a (mai) dilua plictiseala am introdus o parte din predicatul de selecție (*F1.NrFact=1120*) chiar în clauza FROM.

NRFACT
1119
1121
1122

Figura 5.46. Facturile emise în aceeași zi cu 1120

Despre intersecția realizată prin joncțiune am discutat la exemplul 17 din paragraful 4.4.4 - *În ce zile s-au vândut și produsul cu denumirea „Produs 1" și cel cu denumirea „Produs 2"?*

Joncționăm o instanță obținută prin joncțiunea PRODUSE-LINIIFACT-FACTURI, în care DenPr = 'Produs 1', cu o altă instanță a aceleași combinații de tabele, în care DenPr = 'Produs 2'.

```
SELECT DISTINCT F1.DataFact
FROM produse P1
INNER JOIN liniifact LF1 ON P1.CodPr = LF1.CodPr
        INNER JOIN facturi F1 ON LF1.NrFact = F1.NrFact
        INNER JOIN facturi F2 ON F1.DataFact=F2.DataFact
        INNER JOIN liniifact LF2 ON LF2.NrFact = F2.NrFact
        INNER JOIN produse P2 ON LF2.CodPr = P2.CodPr
WHERE P1.DenPr = 'Produs 1' AND P2.DenPr = 'Produs 2'
```

Inserând filtrarea după cele două produse în clauza FROM se obţine o formă uşor schimbată a consultării:

```
SELECT DISTINCT F1.DataFact
FROM produse P1
        INNER JOIN liniifact LF1 ON P1.CodPr = LF1.CodPr
        AND P1.DenPr = 'Produs 1'
        INNER JOIN facturi F1 ON LF1.NrFact = F1.NrFact
        INNER JOIN facturi F2 ON F1.DataFact=F2.DataFact
        INNER JOIN liniifact LF2 ON LF2.NrFact = F2.NrFact
        INNER JOIN produse P2 ON LF2.CodPr = P2.CodPr
        AND P2.DenPr = 'Produs 2'
```

Capitolul 6
Funcții SQL

Am făcut, deja, superficial, cunoștință cu o serie de funcții pentru numere, șiruri și date calendaristice. În acest capitol vom dezvolta subiectul, fără a epuiza, nici pe departe, impresionanta tipologie a funcțiilor disponibile. Cu atât mai mult cu cât portofoliul funcțiilor este destul de eterogen de la un SGBD la altul. Până spre finalul lucrării vom avea în vedere, exclusiv, funcțiile standard (*buit-in functions*), urmând ca în capitolul 16 să acordăm câteva pagini funcțiilor definite de utilizator.

Prima ediție a cărții de față nu a avut un capitol special dedicat funcțiilor SQL. Am apelat la funcțiile-sistem (predefinite) doar în măsura nevoilor de interogare, comentându-le în exemple. Pentru a lămuri cât de cât discuția, trebuie subliniat că funcțiile din primele șase paragrafe (6.1-6.6) se aplică la nivel de linie, asupra valorii unui atribut sau a unei expresii. Unii autori ([Jones *et al.* 2005] p. 33, [Sîrbu 2007] p. 12) le numesc *scalare* prin comparație cu funcțiile-*agregat*, care se calculează la nivel de grup de înregistrări (sau la nivelul tuturor liniilor din tabelă care satisfac o eventuală condiție de filtrare). Adevărul este că și funcțiile-agregat sunt oarecum scalare, întrucât returnează o valoare simplă (să-i zicem, cu strângere de inimă, *atomică*). Dacă ne raportăm la argumentele asupra cărora operează funcțiile, atunci categorizarea este acceptabilă.

După tipul datelor asupra cărora se aplică, putem delimita funcții dedicate numerelor, șirurilor de caractere, datelor calendaristice și intervalelor. La acestea mai putem adăuga funcții de conversie (a numerelor în șiruri de caractere, a șirurilor în date calendaristice etc.), funcții pentru manipularea valorilor NULLe (vezi capitolul 8), funcții pentru colecții (vectori) și funcții-sistem - cele care returnează anumiți parametri ai SGBD-ului, schemei sau sesiunii curente (ora sistemului, utilizatorul curent etc.). Categorizarea se poate face și mai fin, în sensul că, spre exemplu, putem face delimitarea între funcții aplicate asupra șirurilor ce returnează tot șiruri (CONCAT, LTRIM, UPPER) și funcții aplicate asupra șirurilor ce returnează numere (CHARACTER_LENGTH, POSITION). La fel și cu multe funcții destinate datelor și intervalelor calendaristice.

6.1. Funcții pentru numere

Nu are rost, pentru scopul lucrării de față, să detaliem funcțiile matematice de tipul ABS (ce returnează valoarea absolută a unui număr), COS, SIN, TAN, ACOS,

ASIN, ATAN (cosinus, sinus, tangentă, arccosinus, arcsinus, arctangentă) ş.a.m.d.[1]. Funcţii precum ABS (pentru valoarea absolută a unui argument), MOD (pentru restul împărţirii a două numere) - vezi figura 6.1 - au o incidenţă relativ redusă în aplicaţiile pentru afaceri. Iată un exemplu pe sintaxa Oracle:

SELECT ABS(15/7), ABS(-15/7), MOD(15,7), MOD(-15,7) FROM dual

ABS(15/7)	ABS(-15/7)	MOD(15,7)	MOD(-15,7)
2,14285714285714285714285714285714285714	2,14285714285714285714285714285714285714	1	-1

Figura 6.1. Exemplu de folosire a funcţiilor ABS şi MOD (în Oracle SQL Developer)

Diferenţele de la un SGBD la altul pot apărea nu numai în titulatura unor funcţii, după cum vom vedea în acest capitol, ci şi la „precizia" afişării. Aceeaşi interogare executată în PostgreSQL (fireşte, fără *FROM dual*) sau în DB2 (înlocuind DUAL cu *SYSIBM.SYSDUMMY1* sau *SYSIBM.DUAL*) extrage un rezultat ca în figura 6.2[2], în care am fi tentaţi să credem că rezultatul funcţiei ABS este un întreg (INTEGER).

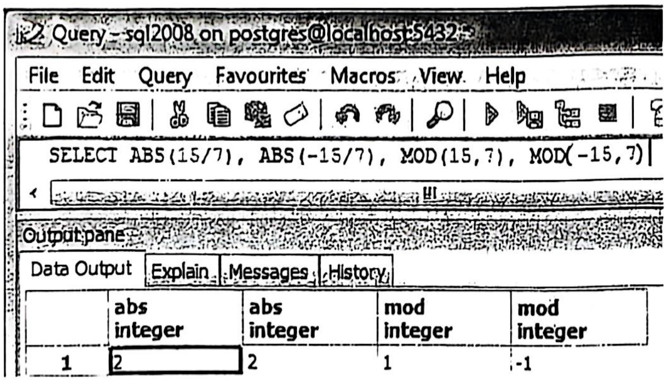


Figura 6.2. Aceeaşi funcţie ABS, precizie diferită în PostgreSQL pgAdmin

În realitate, diferenţa apare de la modul în care PostgreSQL-ul evaluează raportul 15/7. Ca exemplu suplimentar, calculăm procentul „unitar" de TVA pentru fiecare produs vândut în factura 1114, procent unitar care este produsul *ProcTVA * PreţUnit,* aplicând funcţia ABS:

```
SELECT DenPr, Cantitate, PretUnit, ProcTVA,
        PretUnit * ProcTVA AS ProcTVAUnitar1,
        ABS( -PretUnit * ProcTVA) AS ABS_ProcTVAUnitar
FROM liniifact INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE NrFact=1114
```

şi vedem că rezultatul este unul cât se poate de normal - vezi figura 6.3 -, rezultat ce coincide cu acelea obţinute în urma lansării acestei interogări în SQLServer şi DB2.

1. Pentru o tratare detaliată a funcţiilor matematice în SQL, vezi [Jones *et al.* 2005], pp. 69-86.
2. Trebuie să precizăm că fiecare SGBD/utilitar de conectare pune la dispoziţie comenzi pentru configurarea multor parametri de afişare şi interpretare a datelor de tip numeric, date calendaristice etc.

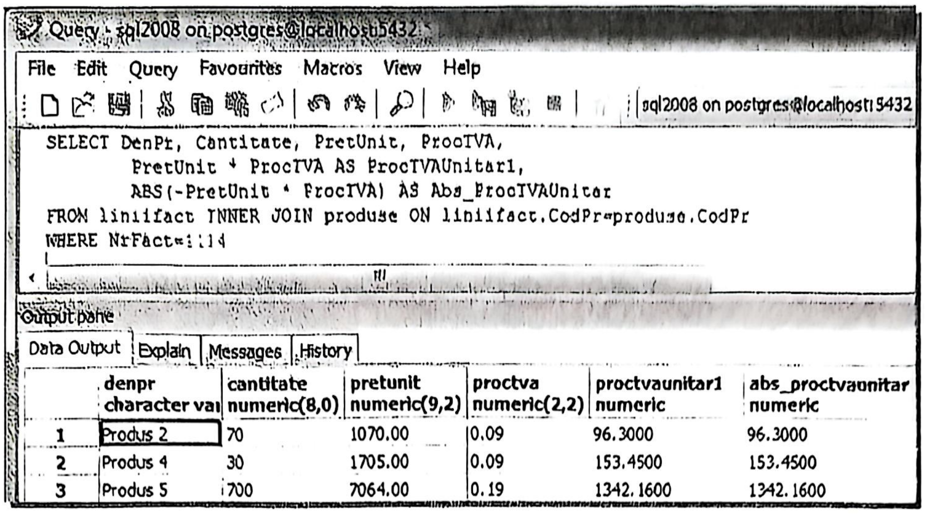


Figura 6.3. Funcția ABS în PostgreSQL pgAdmin

Echivalent funcției *modulo* (MOD), MS SQL Server foloseşte operatorul procent (%), aşa încât prima interogare din acest paragraf poate funcționa în SQL Server doar în forma:

SELECT ABS(15/7), ABS(-15/7), 15 % 7, -15 %7

Tot pentru încălzire, exemplificăm folosirea altor două funcții, POWER (pentru ridicarea la putere) şi SQRT (radical de ordinul 2 sau rădăcină pătrată). Numerele calculate sunt 2^{10}, $2^{0,5}$, 2^{-3} şi $\sqrt{2}$ (ultimul este radical din 2), pe sintaxa Oracle:

SELECT POWER(2,10), POWER (2, 0.5), POWER(2, -3), SQRT(2)
FROM dual

Interogarea funcționează în DB2, MS SQL Server, Oracle, şi PostgreSQL (vezi figura 6.4.), chiar dacă numărul de poziții fracționare afişat este diferit. Funcția POWER este implementată cu numele de POW în MySQL (nume utilizabil şi în PostgreSQL).

DB2:

SELECT POWER(2,10), POWER (2, 0.5), POWER(2, -3), SQRT(2) FROM sysibm.dual

1	2	3	4
1024	1.414	0	1.414

Oracle:

SELECT POWER(2,10), POWER (2, 0.5), POWER(2, -3), SQRT(2) FROM dual

POWER(2,10)	POWER(2,0.5)	POWER(2,-3)	SQRT(2)
1024	1,4142135623730950488016887242096980785 5	0,125	1,4142135623730950488016887242096980785 7

PostgreSQL:

SELECT POWER(2,10), POWER (2, 0.5), POWER(2, -3), SQRT(2)

	power double precis	power numeric	power double precis	sqrt double precis
1	1024	1.414213562 37	0.125	1.41421356237

SQL Server:

SELECT POWER(2,10), POWER (2, 0.5), POWER(2, -3), SQRT(2)

	(No column name)	(No column name)	(No column name)	(No column name)
1	1024	1	0	1.4142135623731

Figura 6.4. Exemplu de folosire a funcțiilor POWER şi SQRT în cele patru servere BD

Probabil că funcţiile numerice cel mai frecvent folosite sunt ROUND, TRUNC, CEILING (sau CEIL) şi FLOOR. Iată un exemplu executat în Oracle, obiectul funcţiilor fiind rezultatul împărţirii lui 7 la 11:

```
SELECT '7/11' , 7/11 AS functie FROM dual
UNION
SELECT 'CEILING(7/11)', CEIL (7/11) FROM dual
UNION
SELECT 'FLOOR(7/11)', FLOOR (7/11) FROM dual
UNION
SELECT 'ROUND(7/11, 2)', ROUND (7/11,2) FROM dual
UNION
SELECT 'TRUNC(7/11, 2)', TRUNC (7/11,2) FROM dual
```

Rezultatul din figura 6.5 arată (dacă mai era nevoie) că: prin CEIL (CEILING) am obţinut cei mai apropiat număr întreg superior lui 0,636363..., adică 1; prin FLOOR am obţinut cel mai apropiat număr întreg mai mic decât 0,636363..., adică 0; iar diferenţa dintre ROUND şi TRUNC este cea dintre rotunjire şi trunchiere (la două poziţii fracţionare).

'7/11'	FUNCTIE
7/11	0,6363636363636363636363636363636363636364
CEILING(7/11)	1
FLOOR(7/11)	0
ROUND(7/11, 2)	0,64
TRUNC(7/11, 2)	0,63

Figura 6.5. Comparaţie între funcţiile CEILING, FLOOR, ROUND şi TRUNC

Din nou PostgreSQL-ul şi DB2-ul fac notă discordantă din cauza modului de evaluare a raportului 7/11 (vezi figura 6.6).

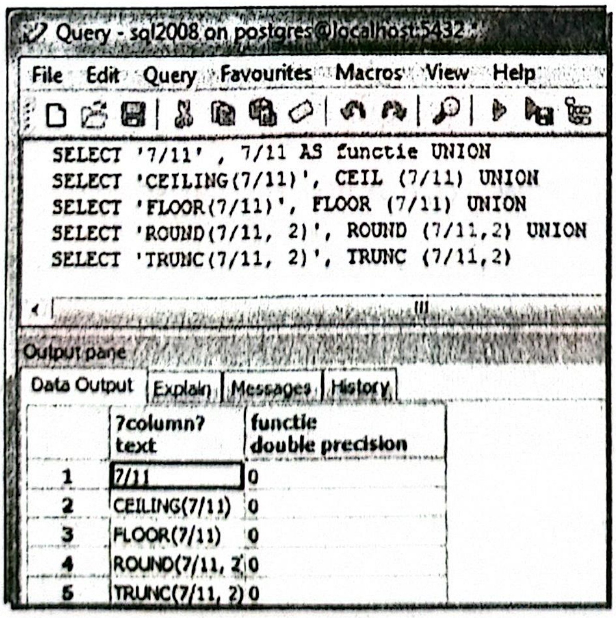


Figura 6.6. Rezultate diferite în PostgreSQL şi DB2 pentru funcţiile CEILING, FLOOR, ROUND şi TRUNC din cauza evaluării fracţiei 7/11

Ne mai liniştim dacă revenim la exemplul legat de calcularea TVA-ului unitar pentru produsele din factura 1114. Rezultatul interogării:

```
SELECT DenPr, Cantitate, PretUnit, ProcTVA,
        PretUnit * ProcTVA AS TVAUnitar1,
        CEIL(PretUnit * ProcTVA) AS TVAUn_CEIL,
        FLOOR(PretUnit * ProcTVA) AS TVAUn_FLOOR,
        TRUNC(PretUnit * ProcTVA,0) AS TVAUn_TRUNC0,
        TRUNC(PretUnit * ProcTVA,1) AS TVAUn_TRUNC1,
        ROUND(PretUnit * ProcTVA,0) AS TVAUn_ROUND0,
        ROUND(PretUnit * ProcTVA,1) AS TVAUn_ROUND1
FROM liniifact INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE NrFact=1114
```

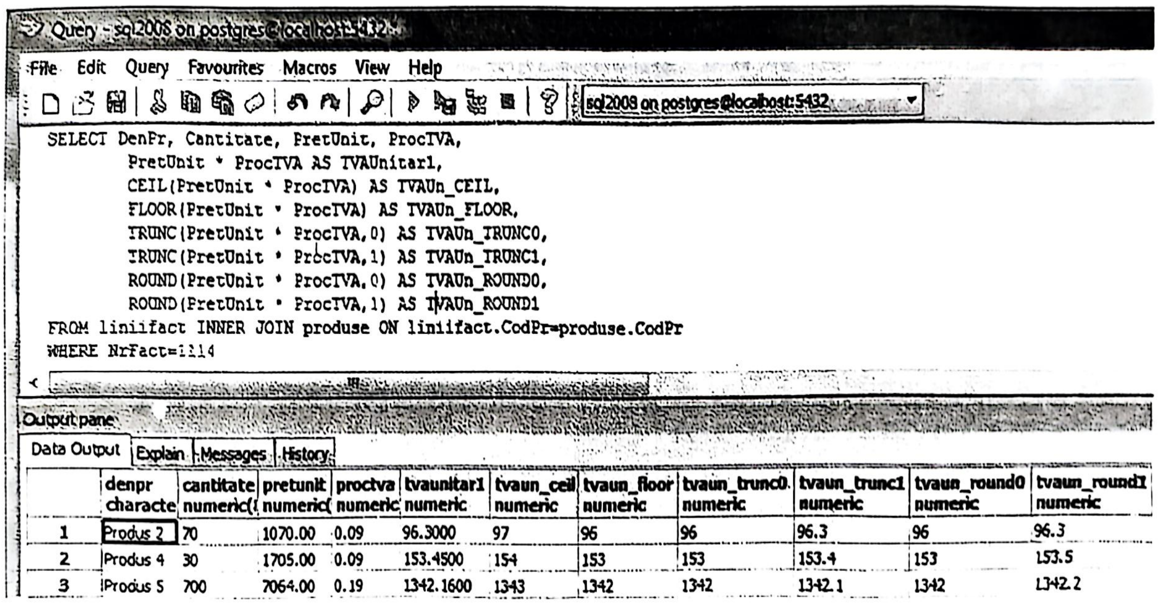


Figura 6.7. Funcțiile CEILING, FLOOR, ROUND şi TRUNC în PostgreSQL

este unul cât se poate de corect – sau cel puțin aşa pare (vezi figura 6.7). Lucrurile stau identic în DB2. SQL Server prezintă două diferențe: în loc de CEIL este obligatoriu să se folosească CEILING, iar trunchierea se face tot cu funcția ROUND, adăugând un al treilea parametru (1):

```
SELECT DenPr, Cantitate, PretUnit, ProcTVA,
        PretUnit * ProcTVA AS TVAUnitar1,
        CEILING(PretUnit * ProcTVA) AS TVAUn_CEILING,
        FLOOR(PretUnit * ProcTVA) AS TVAUn_FLOOR,
        ROUND(PretUnit * ProcTVA,0,1) AS TVAUn_TRUNC0,
        ROUND(PretUnit * ProcTVA,1,1) AS TVAUn_TRUNC1,
        ROUND(PretUnit * ProcTVA,0) AS TVAUn_ROUND0,
        ROUND(PretUnit * ProcTVA,1) AS TVAUn_ROUND1
FROM liniifact INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE NrFact=1114
```

Trunchierea şi rotunjirea se pot face nu numai la poziții fracționare, ci şi la nivel de unități, zeci, sute, mii ş.a.m.d. Înmulțim rezultatul diviziunii 7/11 cu 10000

(adică 10^4), prilej de exemplificare a funcţiei POWER (ridicare la putere), păstrând funcţiile din SELECT-ul anterior (vezi figura 6.8):

```
SELECT '7/11 * 10000' , 7/11 * POWER (10,4) AS functie FROM dual
UNION
SELECT 'CEILING(7/11 * 10000)', CEIL (7/11 * POWER (10,4)) FROM dual
UNION
SELECT 'FLOOR(7/11 * 10000)', FLOOR (7/11 * POWER (10,4)) FROM dual
UNION
SELECT 'ROUND(7/11 * POWER (10,4), -2)', ROUND (7/11 * POWER (10,4), -2)
FROM dual
UNION
SELECT 'TRUNC(7/11 * POWER (10,4), -2)', TRUNC (7/11 * POWER (10,4),-2)
FROM dual
```

'7/11*10000'	FUNCTIE
7 / 11 * 10000	6363,6363636363636363636363636363636363636364
CEILING(7 / 11 * 10000)	6364
FLOOR(7 / 11 * 10000)	6363
ROUND(7/11 * POWER (10,4), -2)	6400
TRUNC(7/11 * POWER (10,4), -2)	6300

Figura 6.8. Rotunjire şi truchiere la nivelul sutelor (Oracle)

După cum vă aşteptaţi, DB2 (interogarea următoare), SQL Server şi PostgreSQL ne furnizează un rezultat eronat (vezi figura 6.9):

```
SELECT '7/11 * 10000' AS Functie, 7/11 * POWER (10,4) AS Rezultat
FROM SYSIBM.SYSDUMMY1
        UNION
SELECT 'CEILING(7/11 * 10000)', CEIL (7/11 * POWER (10,4))
FROM SYSIBM.SYSDUMMY1
        UNION
SELECT 'FLOOR(7/11 * 10000)', FLOOR (7/11 * POWER (10,4))
FROM SYSIBM.SYSDUMMY1
        UNION
SELECT 'ROUND(7/11 * POWER (10,4), -2)', ROUND (7/11 * POWER (10,4), -2)
FROM SYSIBM.SYSDUMMY1
        UNION
SELECT 'TRUNC(7/11 * POWER (10,4), -2)' AS "Functie",
        TRUNC (7/11 * POWER (10,4),-2) AS "Rezultat"
FROM SYSIBM.SYSDUMMY1
```

1	2
7 / 11 * 10000	0
CEILING(7 / 11 * 10000)	0
FLOOR(7 / 11 * 10000)	0
ROUND(7/11 * POWER (10,4), -2)	0
TRUNC(7/11 * POWER (10,4), -2)	0

Figura 6.9. Rotunjire şi truchiere eronate la nivelul sutelor (DB2)

Unde mai punem că cele două coloane nu sunt denumite cum am dori, chiar dacă folosim clauza AS (din cauza UNION-ilor). În schimb, dacă aplicăm funcțiile datelor din tabelele LINIIFACT și PRODUSE (ca în figura 6.7), rezultatul este cât se poate de corect (vezi figura 6.10):

```
SELECT DenPr, Cantitate, PretUnit, ProcTVA,
        PretUnit * ProcTVA AS TVAUnitar1,
        TRUNC(PretUnit * ProcTVA,-1) AS "TVAUn_TRUNC-1",
        ROUND(PretUnit * ProcTVA,-1) AS "TVAUn_ROUND-1",
        TRUNC(PretUnit * ProcTVA,-2) AS "TVAUn_TRUNC-2",
        ROUND(PretUnit * ProcTVA,-2) AS "TVAUn_ROUND-2"
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
WHERE NrFact = 1114
```

DENPR	CANTITATE	PRETUNIT	PROCTVA	TVAUNITAR1	TVAUn_TRUNC-1	TVAUn_ROUND-1	TVAUn_TRUNC-2	TVAUn_ROUND-2
Produs 2	70	1.070,00	0,09	96,3000	90,0000	100,0000	0,0000	100,0000
Produs 4	30	1.705,00	0,09	153,4500	150,0000	150,0000	100,0000	200,0000
Produs 5	700	7.064,00	0,19	1.342,1600	1.340,0000	1.340,0000	1.300,0000	1.300,0000

Figura 6.10. Rotunjire şi truchiere corecte la nivelul sutelor (DB2)

Acelaşi rezultat îl obținem în MS SQL Server prin interogarea:

```
SELECT DenPr, Cantitate, PretUnit, ProcTVA,
        PretUnit * ProcTVA AS TVAUnitar1,
        ROUND(PretUnit * ProcTVA,-1,1) AS [TVAUn_TRUNC-1],
        ROUND(PretUnit * ProcTVA,-1) AS [TVAUn_ROUND-1],
        ROUND(PretUnit * ProcTVA,-2,1) AS [TVAUn_TRUNC-2],
        ROUND(PretUnit * ProcTVA,-2) AS [TVAUn_ROUND-2]
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
WHERE NrFact = 1114
```

6.2. Funcții pentru şiruri de caractere

O bună parte dintre cele mai utile funcții pentru şiruri de caractere au fost deja folosite în capitolele 3 şi 5 la definirea unor restricții, inserări de linii şi interogări. Cele mai folosite din această categorie sunt funcțiile CONCAT, LOWER, UPPER, LENGTH, SUBSTR, TRIM şi REPLACE[3].

Despre CONCAT se poate spune că, în afara acțiunii sale (concatenarea a două sau mai multe şiruri de caractere), prezintă uneori (în Oracle şi DB2) limitarea, destul de apăsătoare, de a avea numai două argumente. De exemplu, a patra interogare din paragraful 5.3 (cu rezultatul în figura 5.9) se poate scrie în Oracle/DB2 astfel:

```
SELECT CONCAT ('Judetul ', Judet) || CONCAT (' se afla in ', Regiune)
        AS Exemplu_Concatenare
FROM judete
```

3. Există diferențe apreciabile în enumerarea funcțiilor pentru şiruri de caractere. Astfel, dacă în [Melton & Simon 2002], pp. 147-151, sunt prezentate: SUBSTRING, UPPER, LOWER, TRIM, TRANSLATE, CONVERT şi OVERLAY, în [Jones et al. 2005], pp. 65-69, sunt descrise: ASCII, CHR (CHAR), CONCAT, LOWER, UPPER, LENGTH (LEN) şi REPLACE. Noi le-am inclus şi pe cele care operează asupra şirurilor, dar returnează numere.

Alte dialecte (de exemplu, MySQL) sunt mai darnice, acceptând oricâte argumente pentru CONCAT[4], în timp ce PostgreSQL şi MS SQL Server nu au implementată această funcţie.

Funcţiile LOWER şi UPPER, implementate în toate cele patru servere BD, au fost întrebuinţate serios chiar de la crearea tabelelor (atunci când doream ca valorile unui atribut să fie introduse numai cu litere mici sau majuscule). Iată un exemplu în care apare şi o funcţie mai puţin standardizată, şi anume cea care transformă iniţiala fiecărui cuvânt din şir în majusculă, restul literelor din fiecare cuvânt fiind transformate în litere mici. În Oracle şi PostgreSQL funcţia este INITCAP, dar în alte dialecte poate avea nume diferit (de exemplu, în Visual FoxPro este PROPER), iar în altele (de exemplu, DB2, MS SQL Server) nu este deloc. Rezultatul din Oracle este cel din figura 6.11.

```
SELECT '  Sirul initial', ' sIr DE TeSt' AS sir_Initial FROM dual
        UNION
SELECT '  MAJUSCULE', UPPER(' sIr DE TeSt') FROM dual
        UNION
SELECT ' litere mici', LOWER(' sIr DE TeSt') FROM dual
        UNION
SELECT 'Initiala (initiale) majuscula. Restul, litere mici', INITCAP(' sIr DE TeSt')
FROM dual
```

'SIRULINITIAL'	SIR_INITIAL
Sirul initial	sIr DE TeSt
MAJUSCULE	SIR DE TEST
litere mici	sir de test
Initiala (initiale) majuscula. Restul, litere mici	Sir De Test

Figura 6.11. UPPER, LOWER, INITCAP

Tot în capitolul 3 am folosit şi funcţii de eliminare a spaţiilor dintr-un şir de caractere - TRIM. Pentru a scăpa de spaţiile de la începutul şirului foloseam LTRIM (de la LEFT TRIM), iar pentru a le elimina pe cele de la sfârşitul şirului foloseam RTRIM (RIGHT TRIM).

Standardele SQL:1999-2008 conţin doar funcţia TRIM cu argumentele LEADING, TRAILING şi BOTH pentru a indică modul în care se realizează „tunsul" spaţiilor. Interogarea următoare a fost executată în Oracle, de aici *FROM dual*-ul de rigoare:

```
SELECT '1: Sirul initial', '{' || '  sIr DE TeSt   ' || '}' AS sir_Initial
FROM dual
        UNION
SELECT '2: TRIM - LEADING', '{' ||
        TRIM(LEADING ' ' FROM('  sIr DE TeSt   ')) || '}'
FROM dual
        UNION
SELECT '3: TRIM - TRAILING', '{' ||
```

4. În schimb, MySQL nu acceptă concatenarea cu operatorul || (sau +).

```
        TRIM(TRAILING ' ' FROM('  sIr DE TeSt   ')) || '}'
FROM dual
        UNION
SELECT '4: TRIM - BOTH', '{' || TRIM( BOTH ' ' FROM ('  sIr DE TeSt   '))
        || '}'
FROM dual
```

Pentru o mai bună vizualizare a spațiilor de la începutul și sfârșitul șirului (vezi figura 6.12), am folosit acoladele. Funcțiile operează similar în DB2 și PostgreSQL, nu însă și în MS SQL Server.

Results:

	'1:SIRULINITIAL'	SIR_INITIAL
1	1: Sirul initial	{ sIr DE TeSt }
2	2: TRIM - LEADING	{sIr DE TeSt }
3	3: TRIM - TRAILING	{ sIr DE TeSt}
4	4: TRIM - BOTH	{sIr DE TeSt}

Figura 6.12. Funcția TRIM

În majoritatea SGBD-urilor, de ani buni se folosesc funcțiile la care am apelat și noi în capitolul 3: LTRIM și RTRIM, tăierea la ambele capete fiind realizată fie prin TRIM (de exemplu, Oracle, PostgreSQL), ALLTRIM (FoxPro) sau LTRIM(RTRIM(...)). Iată transcrierea în MS SQL Server a interogării de mai sus:

```
SELECT '1: Sirul initial', '{' + '  sIr DE TeSt   ' + '}' AS sir_Initial
       UNION
SELECT '2: TRIM - LEADING', '{' + LTRIM('  sIr DE TeSt   ') + '}'
       UNION
SELECT '3: TRIM - TRAILING', '{' + RTRIM('  sIr DE TeSt   ') + '}'
       UNION
SELECT '4: TRIM - BOTH', '{' + LTRIM(RTRIM('  sIr DE TeSt   ')) + '}'
```

Cel mai frecvent, funcția se folosește pentru eliminarea spațiilor. Însă, după cum probabil ați observat, după LEADING, TRAILING și BOTH, spațiul a fost indicat explicit prin ' '. Putem specifica orice caracter în locul spațiului. De exemplu, **TRIM(BOTH 'E' FROM 'ESCHIVARE')** va returna **'SCHIVAR'**.

Dacă funcția TRIM elimină spații (sau un alt caracter), multe dialecte SQL au o funcție opusă - PAD care „umple" cu spații sau cu un alt caracter un șir de caractere până când acesta (șirul) atinge o anumită lungime. Umplerea se poate face la stânga (LPAD, adică LEFT PAD) sau la dreapta (RPAD, de la RIGHT PAD). Astfel, dacă în tabela CLIENȚI dorim să afișăm toate valorile atributului Adresa pe 30 caractere și, acolo unde lungimea este mai mică decât 30 să se umple cu asteriscurile necesare, putem folosi funcțiile RPAD și LPAD, al căror efect este vizibil în figura 6.13:

```
SELECT DenCl, Adresa,
        RPAD(adresa, 30, '*') AS Exemplu_RPAD,
        LPAD(adresa, 30, '*') AS Exemplu_LPAD
FROM clienti
```

DENCL	ADRESA	EXEMPLU_RPAD	EXEMPLU_LPAD
Client 1 SRL	Tranzitiei, 13 bis	Tranzitiei, 13 bis************	************Tranzitiei, 13 bis
Client 2 SA	(null)	(null)	(null)
Client 3 SRL	Prosperitatii, 22	Prosperitatii, 22*************	*************Prosperitatii, 22
Client 4	Sapientei, 56	Sapientei, 56*****************	*****************Sapientei, 56
Client 5 SRL	(null)	(null)	(null)
Client 6 SA	Pacientei, 33	Pacientei, 33*****************	*****************Pacientei, 33
Client 7 SRL	Victoria Capitalismului, 2	Victoria Capitalismului, 2****	****Victoria Capitalismului, 2

Figura 6.13. Funcţii PAD

Din păcate, cei care se aşteptau ca alinierea să fie perfectă (la urma urmei, am stabilit limita de reprezentare la 30 de caractere) au toate motivele să fie dezamăgiţi, întrucât în mediile grafice (cum este Oracle SQL Developer sau PostgreSQL pgAdmin) spaţiul necesar reprezentării unui caracter diferă în funcţie de forma acestuia. În vederea evitării acestui tip de probleme, MS SQL Server-ul şi DB2 nu au implementat o asemenea funcţie.

Lungimea unui şir de caractere poate fi determinată în standardul SQL prin funcţia CHARACTER_LENGTH sau „prescurtarea" CHAR_LENGTH ([Melton & Simon 2002], pp. 144-145). SGBD-urile sunt mai pragmatice, funcţia fiind, simplu, LENGTH (în toate cele patru servere) sau LEN (MS SQL Server). Şi în privinţa funcţiei pentru extragerea unei porţiuni dintr-un şir de caractere - SUBSTRING - apar diferenţe între standard şi implementări. În stardardele post-SQL:1999, pentru a extrage primele patru şi ultimele patru caractere dintr-un şir (vezi figura 6.14) sintaxa ar fi:

```
SELECT 'sIr DE TeSt' AS sir,
        CHARACTER_LENGTH('sIr DE TeSt') AS lungime,
        SUBSTRING('sIr DE TeSt' FROM 1 FOR 4) AS Primele_patru,
        SUBSTRING('sIr DE TeSt' FROM
                    CHARACTER_LENGTH('sIr DE TeSt')-3 FOR 4)
        AS Ultimele_patru
```

Results:

	SIR	LUNGIME	PRIMELE_PATRU	ULTIMELE_PATRU
1	sIr DE TeSt	11	sIr	TeSt

Figura 6.14. Funcţiile CHARACTER_LENGTH şi SUBSTRING

DB2, PostgreSQL şi Oracle prezintă funcţia SUBSTR cu trei argumente: şirul propriu-zis, poziţia de pe care începe extragerea şi numărul de caractere extras:

```
SELECT 'sIr DE TeSt' AS sir, LENGTH('sIr DE TeSt') AS lungime,
        SUBSTR('sIr DE TeSt',1,4) AS Primele_patru,
        SUBSTR('sIr DE TeSt', LENGTH('sIr DE TeSt')-3,4) AS Ultimele_patru
```
[5]

5. Interogarea funcţionează în PostgreSQL. Fireşte, în Oracle se adaugă *FROM dual*, iar în DB2 *FROM sysibm.dual*.

MS SQL Server folosește LEN pe post de LENGTH și SUBSTRING pe post de SUBSTR. De asemenea, dacă porțiunea extrasă se află la începutul sau la sfârșitul șirului de caractere, se pot folosi, în loc de SUBSTRING, funcțiile LEFT și RIGHT:

```
SELECT 'sIr DE TeSt' AS sir, LEN('sIr DE TeSt') AS lungime,
        LEFT('sIr DE TeSt',4) AS Primele_patru,
        RIGHT('sIr DE TeSt', 4) AS Ultimele_patru
```

Diferențe majore apar și în cazul funcției care caută apariția unui șir de caractere - șir1 - în alt șir (tot de caractere) - șir2 - și returnează poziția pe care apare șir1 în șir2 sau 0, când căutarea este fără succes. În SQL:1999 apare funcția POSITION. Spre exemplu, *POSITION ('T' IN 'sIr DE TeSt')* returnează 8, pentru că T apare pe a opta poziție din *'sIr DE TeSt'*. SGBD-urile, însă, dispun de funcții echivalente, cum este INSTR - ceva mai elegantă, deoarece permite căutarea începând cu o poziție specificată și, în plus, caută a *n*-a apariție a lui *șir1* în *șir2*. Exemplul următor (executat în Oracle) caută și afișează prima, a doua și a treia apariție a literei **e** în **sir de test**, precum și prima apariție a succesiunii **sirul** în același **sir de test**. Căutarea se face începând cu primul caracter din *sir de test*.

```
SELECT 'sir de test' AS sir,
        INSTR('sir de test','e',1, 1) AS Poz1_e,
        INSTR('sir de test','e',1, 2) AS Poz2_e,
        INSTR('sir de test','e',1, 3) AS Poz3_e,
        INSTR('sir de test','sirul',1) AS Poz1_sirul
FROM dual
```

După cum arată și rezultatul din figura 6.15, atunci când nu există o a *n*-a apariție a lui *șir1* în *șir2* (a treia apariție a lui e, prima apariție a lui *sirul*, rezultatul este zero.

Results:

	SIR	POZ1_E	POZ2_E	POZ3_E	POZ1_SIRUL
1	sir de test	6	9	0	0

Figura 6.15. Funcția INSTR (echivalentă funcției POSITION)

În PostgreSQL se poate determina doar prima apariție a unui subșir într-un șir, cu ajutorul funcției POSITION:

```
SELECT 'sir de test' AS sir, POSITION('e' IN 'sir de test') AS Poz1_e
```

în DB se poate folosi și POSSTR și LOCATE:

```
SELECT 'sir de test' AS sir,
        POSSTR('sir de test', 'e') AS Poz1_e,
        LOCATE('e', 'sir de test') AS Poz1_e_v2
FROM SYSIBM.SYSDUMMY1
```

iar în MS SQL Server funcția se numește PATINDEX și folosește „joker"-ul % pe care l-am folosit în paragraful 5.6:

```
SELECT PATINDEX('%e%', 'sir de test')
```

Și dacă tot am atacat particularități ale funcțiilor de căutare în șiruri de caractere, nu ar fi corect să nu amintim funcția Oracle REGEXP_INSTR și „vecinele" sale, REGEXP_LIKE, REGEXP_COUNT, REGEXP_REPLACE și REGEXP_SUBSTR.

Acest grup de funcţii poate fi, la fel de bine, legat de paragraful 5.6. Oracle nu a implementat predicatul SIMILAR TO dedicat aşa-numitelor *regular expressions*, însă îşi „recuperează" onoarea cu cele cinci funcţii. Iată o variantă ce foloseşte funcţia REGEXP_LIKE pentru a extrage din tabela PERSOANE liniile în care numele începe cu litera I:

```
SELECT *
FROM persoane
WHERE REGEXP_LIKE(Prenume, 'I')
```

Partea cea mai interesantă o constituie mulţimea operatorilor şi metasimbolurilor ce pot fi utilizate pentru construirea măştii de căutare. Spre exemplu, dacă dorim să extragem linile în care prenumele este Ion, Ioan sau Ioana, şablonul este:

```
SELECT *
FROM persoane
WHERE REGEXP_LIKE(Prenume, '^I(o|oa)n(|a)$')
```

Simbolurile ^ şi $ marchează începutul şi sfârşitul şirului, dar pot preciza şi „capetele" unei linii dintr-un şir care se întinde pe mai multe rânduri. Bara verticală introduce o structură alternativă, iar parantezele grupează expresii.

Reluăm unul dintre exemplele din paragraful 5.6 (figura 5.33): *Ce persoane stau într-un bloc al cărui nume este alcătuit dintr-o literă şi o cifră* (de exemplu, H2, B5, A9)? Varianta Oracle de mai jos este mai bună decât cea din capitolul 5, întrucât SELECT-ul respectiv ar fi extras şi blocurile al căror nume era alcătuit din două cifre:

```
SELECT *
FROM persoane
WHERE REGEXP_LIKE(Adresa, 'Bl.\w\d')
```

Caracterele „ *w*" substituie o literă, în timp ce „ *d*" o cifră. La fel de corectă este varianta în care o literă este substituită prin intervalul [A-Z][6], iar o cifră substituită prin intervalul [0-9]:

```
SELECT * FROM persoane WHERE REGEXP_LIKE(Adresa, 'Bl.[A-Z][0-9]')
```

Există foarte multe modalităţi de construire a expresiilor de căutare. Eu mă opresc aici, sperând că v-am trezit cât de cât curiozitatea ca sa încercaţi ceva şi pe cont propriu.

Am văzut în paragraful 5.4 că ordonarea se aplică nu numai la numere şi date calendaristice, ci şi la şiruri de caractere. Elementul luat în calcul la ordonare este codul ASCII sub care este stocat orice caracter într-un calculator. Comanda următoare afişează (în Oracle) caracterele ce corespund codurilor ASCII cu numerele 40, 41, 49, 50, 65 şi 97 (vezi figura 6.16).

```
SELECT 40 AS argument, CHR(40) AS "CHR(argument)" FROM dual
UNION SELECT 41, CHR(41) FROM dual
UNION SELECT 49, CHR(49) FROM dual
UNION SELECT 50, CHR(50) FROM dual
```

6. Literele din denumirea blocului se scriu întotdeauna cu majuscule.

```
UNION SELECT 65, CHR(65) FROM dual
UNION SELECT 97, CHR(97) FROM dual
```

ARGUMENT	CHR(argument)
40	(
41	)
49	1
50	2
65	A
97	a

Figura 6.16. Funcția CHR

Dacă eliminăm *FROM dual*, sintaxa este acceptată de PostgreSQL, iar înlocuind *FROM dual* cu *FROM sysibm.sysdummy1* (sau cu *FROM sysibm.dual*) interogarea poate fi executată în DB2. În MS SQL Server, interogarea este similară cu PostgreSQL, doar că numele funcției e altul, respectiv CHAR.

Funcția inversă, care returnează codul ASCII al unui caracter dat este chiar ASCII. Despre funcții precum TRANSLATE/REPLACE, CONVERT sau OVERLAY și corespondentele lor în diverse dialecte nu vom discuta în această lucrare[7].

6.3. Funcții pentru date calendaristice

Din SQL:1999 a fost introdusă funcția EXTRACT prin care putem afla componentele unei date/unui moment calendaristic(e). Funcția este implementată în Oracle și PostgreSQL, dar nu și în DB2 și MS SQL Server. Cele mai întrebuințate argumente sunt YEAR, MONTH și DAY. Iată un format simplu al unei interogări ce extrage, pe rând, anul, luna și ziua dintr-o dată calendaristică, interogare executată în Oracle și al cărei rezultat este cel din figura 6.17.

```
SELECT DATE'2007-09-17',
        EXTRACT (YEAR FROM DATE'2007-09-17') AS anul,
        EXTRACT (MONTH FROM DATE'2007-09-17') AS luna,
        EXTRACT (DAY FROM DATE'2007-09-17') AS Ziua
FROM dual
```

DATE'2007-09-17'	ANUL	LUNA	ZIUA
17-09-2007	2007	9	17

Figura 6.17. Funcții EXTRACT aplicate asupra unei date calendaristice (Oracle)

EXTRACT poate fi aplicată și asupra unor literali/atribute/variabile de tip dată-timp[8] (TIMESTAMP), ca în exemplul următor (vezi figura 6.18):

```
SELECT TIMESTAMP'2007-09-17 18:30:21',
EXTRACT (DAY FROM TIMESTAMP'2007-09-17 18:30:21') AS Ziua,
EXTRACT (HOUR FROM TIMESTAMP'2007-09-17 18:30:21') AS Ora,
```

7. Pentru detalii, vezi [Melton & Simon 2002], [Jones *et al.* 2005].
8. Pentru tipul dată-timp (TIMPSTAMP), vom folosi, un pic forțat, termenul *moment*.

```
EXTRACT (MINUTE FROM TIMESTAMP'2007-09-17 18:30:21') AS Minutele,
EXTRACT (SECOND FROM TIMESTAMP'2007-09-17 18:30:21') AS Secundele
FROM dual
```

TIMESTAMP'2007-09-1718:30:21'	ZIUA	ORA	MINUTELE	SECUNDELE
17-09-2007 18:30:21,000000000	17	18	30	21

Figura 6.18. Funcţia EXTRACT aplicată asupra unui literal TIMESTAMP

PostgreSQL-ul este mai generos, cu funcţia EXTRACT fiind posibilă aflarea şi a altor informaţii dintr-o dată sau dintr-un moment: secolul (CENTURY), deceniul (DECADE), ziua săptămânii (DOW - Day Of the Week), ziua din an (DOY - Day Of the Year), săptămâna din an (WEEK), trimestrul (QUARTER) etc. Iată asemenea informaţii pentru momentul *11 noiembrie 2007, ora 23:55:10*:

```
SELECT 'Momentul:' AS Parametru, ' ' || TIMESTAMP'2007-11-10 23:55:10' AS Valoare
    UNION
SELECT 'Secol:', ' ' || EXTRACT (CENTURY FROM TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Deceniu:', ' ' || EXTRACT (DECADE FROM TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Zi din sapt.:', ' ' || EXTRACT (DOW FROM TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Zi din an:', ' ' || EXTRACT (DOY FROM TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Sapt. din an:', ' ' || EXTRACT (WEEK FROM TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Trimestru din an:', ' ' || EXTRACT (QUARTER FROM
    TIMESTAMP'2007-11-10 23:55:10')
```

Pentru a asigura afişarea pe mai multe linii (figura 6.19) am folosit operatorul UNION. Cum, însă, prima linie avea pe a doua coloană o valoare TIMESTAMP, iar celelalte valori NUMERIC, a trebuit să forţăm conversia lor în şiruri de caractere prin concatenarea cu un spaţiu.

parametru text	valoare text
Momentul:	2007-11-10 23:55:10
Deceniu:	200
Sapt. din an:	45
Secol:	21
Trimestru din an:	4
Zi din an:	314
Zi din sapt.:	6

Figura 6.19. Generozităţi PostgreSQL în ceea ce priveşte funcţia EXTRACT

Aflarea altor informaţii, precum în ce zi a săptămânii cade o anumită dată presupune folosirea unor funcţii specifice fiecărui SGBD, cum ar fi TO_CHAR în Oracle şi PostgreSQL, DOW/CDOW în Visual FoxPro, DAYOFWEEK în DB2 etc. PostgreSQL-ul însă este destul de elegant, deoarece ziua săptămânii poate fi aflată

tot cu EXTRACT (DOW...), valoarea returnată fiind 0 pentru duminică, 1 pentru luni, ... 6 pentru sâmbătă. Unde mai punem că PostgreSQL poate obține asemenea informații și printr-o funcție proprietară - DATE_PART:

```
SELECT 'Momentul:' AS Parametru, ' ' || TIMESTAMP'2007-11-10 23:55:10' AS Valoare
    UNION
SELECT 'Secol:', ' ' || DATE_PART ('CENTURY', TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Deceniu:', ' ' || DATE_PART ('DECADE', TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Zi din sapt.:', ' ' || DATE_PART ('DOW', TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Zi din an:', ' ' || DATE_PART ('DOY', TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Sapt. din an:', ' ' || DATE_PART ('WEEK', TIMESTAMP'2007-11-10 23:55:10')
    UNION
SELECT 'Trim. din an:', ' ' || DATE_PART ('QUARTER', TIMESTAMP'2007-11-10 23:55:10')
```

Interesant este și faptul că, în Oracle, o parte din funcțiile pe care noi le-am încadrat la matematice, TRUNC și ROUND, sunt utile și pentru date calendaristice (vezi figura 6.20):

```
SELECT DATE'2007-09-17',
TRUNC(DATE'2007-09-17','MONTH') AS trunchiere_data,
ROUND(DATE'2007-09-17','MONTH') AS rotunjire_data
FROM dual
```

DATE'2007-09-17'	TRUNCHIERE_DATA	ROTUNJIRE_DATA
17-09-2007	01-09-2007	01-10-2007

Figura 6.20. Trunchierea și rotunjirea unei date calendaristice la nivel de lună (Oracle)

Pentru PostgreSQL avem două vești rele și două bune. Prima (rea) este că nu putem folosi funcțiile ROUND și TRUNC pentru date calendaristice. A doua (bună) este că dispunem de funcția DATE_TRUNC care se comportă similar funcției TRUNC din interogarea anterioară. Din păcate, nu avem și un DATE_ROUND, iar DATE_TRUNC poate folosi argumente inacceptabile în Oracle (de exemplu, WEEK):

```
SELECT DATE'2007-09-19',
        DATE_TRUNC('MONTH', DATE'2007-09-19') AS trunchiere_luna,
        DATE_TRUNC('WEEK', DATE'2007-09-19') AS trunchiere_sapt
```

19 septembrie 2007 a picat într-o miercuri, așa că trunchierea la nivel de săptămână a returnat data precedentei zile de luni (care marchează începutul săptămânii) - vezi figura 6.21.

date date	trunchiere_luna timestamp with time zone	trunchiere_sapt timestamp with time zone
2007-09-19	2007-09-01 00:00:00+03	2007-09-17 00:00:00+03

Figura 6.21. Trunchieri ale unei date calendaristice la nivel de lună și săptămână (PostgreSQL)

În MS SQL Server, veştile sunt la fel de amestecate. Cea mai rea este că nu există funcţia EXTRACT, şi nu e prima dată când luăm cunoştinţă de distanţa produsului faţă de standardele SQL. Totodată, însă, cea mai mare parte a informaţiilor aflate prin funcţia EXTRACT poate fi obţinută cu ajutorul câtorva funcţii proprietare. Conţinutul din figura 6.17 se obţine prin funcţiile YEAR, MONTH şi DAY:

```
SELECT '2007-09-17',
        YEAR('2007-09-17') AS anul,
        MONTH ('2007-09-17') AS luna,
        DAY ('2007-09-17') AS Ziua
```

dar şi cu mai generala funcţie DATEPART care seamănă binişor cu „surata" sa din PostgreSQL:

```
SELECT '2007-09-17',
        DATEPART(YEAR, '2007-09-17') AS anul,
        DATEPART(MONTH, '2007-09-17') AS luna,
        DATEPART(DAY, '2007-09-17') AS Ziua
```

Tot funcţia DATEPART ne garantează şi informaţiile obţinute (în Oracle) în figura 6.18:

```
SELECT '2007-09-17',
        DATEPART(DAY, '2007-09-17 18:30:21') AS Ziua,
        DATEPART(HH, '2007-09-17 18:30:21') AS Ora,
        DATEPART(MI, '2007-09-17 18:30:21') AS Minutele,
        DATEPART(SS, '2007-09-17 18:30:21') AS Secundele
```

Alte informaţii extrase dintr-o dată calendaristică cu ajutorul funcţiei DATEPART trebuie să folosească argumentele:

- QUARTER (sau QQ sau Q) pentru trimestru;
- DAYOFYEAR (sau DY sau Y) pentru ziua din an;
- WEEK (sau WK sau WW) pentru săpămână;
- WEEKDAY (sau DW) pentru ziua din săpămână;
- DAYOFYEAR (sau DY sau Y) pentru ziua din an.

Pentru a vedea cu ce ziuă începe săptămâna (prima zi a săptămânii) se poate folosi:

```
SELECT @@DATEFIRST
```

fiind returnat un număr cuprins între 1 (pentru luni) şi 7 (pentru duminică). Prima zi a săptămânii poate fi modificată prin comanda SET DATEFIRST, dar lucrul acesta nu prea ne interesează acum.

În DB2, calificarea unei constante de tip dată calendaristică s-a realizat până acum prin încadrarea datei (în format ISO) între apostrofuri. Acelaşi lucru se poate realiza şi prin funcţia DATE:

```
SELECT '2007-09-17' AS O_Data, DATE('2007-09-17') AS Aceeasi_Data
FROM sysibm.sysdummy1
```

Ca şi în SQL Server, în DB2 există funcţii speciale pentru extragerea zilei, lunii, anului dintr-o dată calendaristică şi, pe lângă acestea, a orei, minutelor şi secundelor dintr-un moment:

```
SELECT CURRENT_TIMESTAMP,
```

```
        YEAR(CURRENT_TIMESTAMP) AS anul,
        MONTH(CURRENT_TIMESTAMP) AS luna,
        DAY(CURRENT_TIMESTAMP) AS Ziua,
        HOUR(CURRENT_TIMESTAMP) AS Ora,
        MINUTE(CURRENT_TIMESTAMP) AS Min,
        SECOND(CURRENT_TIMESTAMP) AS Sec
FROM sysibm.sysdummy1
```

La acestea se mai adaugă:

- DAYNAME - pentru numele zilei din săptămână;
- DAYOFWEEK - pentru numărul zilei din săptămână (1 - duminică, ..., 7 - sâmbătă) sau DAYOFWEEK_ISO pentru numărul standard (ISO) al zilei din săptămână (1 - luni, ..., 7 - duminică);
- DAYOFYEAR - pentru ziua din an (numărul său);
- WEEK pentru săptămână (numărul său în cadrul anului), în condițiile în care prima zi a săptămânii este considerată duminică; WEEK_ISO funcționează identic, doar că prima zi a săptămânii este luni;
- MONTHNAME - pentru numele lunii;
- QUARTER - pentru trimestru;
- MICROSECONDS - pentru microsecundele dintr-un moment (TIMESTAMP).

6.4. Funcții pentru intervale

În paragraful 5.3, pentru a exemplifica câteva expresii cu date calendaristice, am folosit tipuri de date INTERVAL dedicate, după cum le spune şi numele, exprimării intervalelor calendaristice. Mai toate SGBD-urile au însă funcții cu vechi state de serviciu în acest sens. Astfel, pentru a specifica intervale calendaristice de ordinul lunilor, în Oracle există funcția ADD_MONTHS, în Visual FoxPro GOMONTH, în SQL Server DATEADD etc.

În Oracle, comanda:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DataFact + INTERVAL '2' MONTH AS Scadenta_Incasare
FROM facturi
```

este echivalentă cu:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        ADD_MONTHS(DataFact,2) AS Scadenta_Incasare
FROM facturi
```

În MS SQL Server, acelaşi rezultat se obține folosind funcția DATEADD astfel:

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DATEADD(MONTH, 2, DataFact) AS Scadenta_Incasare
FROM facturi
```

Cu DATEADD suntem în stare să lucrăm cu orice gen de intervale pe care le-am ocolit în SQL Server până acum. Revenim la o problemă formulată în

paragraful 5.3 – *dorim să afişăm pentru fiecare factură ce dată va fi peste un an, două luni şi 25 de zile de la momentul emiterii.*

```
SELECT NrFact AS Factura, DataFact AS Data_Facturare,
        DATEADD(MONTH, 14, DataFact) + 25
AS O_Data_Viitoare
FROM facturi
```

Pentru aflarea numărului de luni dintre două date calendaristice, înainte de apariţia tipurilor de date INTERVAL, în Oracle se putea folosi doar funcţia MONTHS_BETWEEN. Reluăm una dintre interogările din paragraful 5.3 în a cărei clauză SELECT adăugăm un atribut calculat pe baza noii funcţii:

```
SELECT NrFact, DATE'2007-11-22', DataFact,
        DATE'2007-11-22' - DataFact AS Zile_Scurse,
        (DATE'2007-11-22' - DataFact) YEAR TO MONTH
                    AS AniLuni_Sc1,
        MONTHS_BETWEEN(DATE'2007-11-22', DataFact)
                    AS Luni_Scurse
FROM facturi
```

După cum se observă în figura 6.22, în lipsa unei funcţii de rotunjire sau trunchiere, MONTHS_BETWEEN este din cale-afară de scrupuloasă cu zecimalele.

NRFACT	DATE'2007-11-22'	DATAFACT	ZILE_SCURSE	ANILUNI_SC1	LUNI_SCURSE
1111	22-11-2007	01-08-2007	113	0-4	3,677419354838709677419354838709677419355
1112	22-11-2007	01-08-2007	113	0-4	3,677419354838709677419354838709677419355
1113	22-11-2007	01-08-2007	113	0-4	3,677419354838709677419354838709677419355
1114	22-11-2007	01-08-2007	113	0-4	3,677419354838709677419354838709677419355
1115	22-11-2007	02-08-2007	112	0-4	3,645161290322580645161290322580645161290
1116	22-11-2007	02-08-2007	112	0-4	3,645161290322580645161290322580645161290
1117	22-11-2007	03-08-2007	111	0-4	3,612903225806451612903225806451612903226
1118	22-11-2007	04-08-2007	110	0-4	3,580645161290322580645161290322580645161
1119	22-11-2007	07-08-2007	107	0-4	3,483870967741935483870967741935483870968

Figura 6.22. Funcţia Oracle MONTHS_BETWEEN

Spre deliciul utilizatorilor (adică al nostru), funcţia EXTRACT poate fi aplicată nu numai asupra unei date, ci şi asupra unui interval de timp. Astfel, pentru a calcula numărul de luni dintre două date putem recurge la soluţii precum cele din interogarea:

```
SELECT NrFact,
        DATE'2008-11-22', DataFact,
        DATE'2008-11-22' - DataFact AS Zile_Scurse,
        (DATE'2008-11-22' - DataFact) YEAR TO MONTH AS AniLuni_Sc,
 EXTRACT (YEAR FROM (DATE'2008-11-22' - DataFact) YEAR TO MONTH) * 12 +
                              EXTRACT (MONTH FROM (DATE'2008-11-22'
                     - DataFact) YEAR TO MONTH) AS Luni_Sc1,
        TRUNC(MONTHS_BETWEEN (DATE'2008-11-22', DataFact),2) AS Luni_Sc2
FROM facturi
```

O parte din liniile rezultatului sunt prezentate în figura 6.23.

PostgreSQL-ul nu este atât de generos la exprimarea directă a diferenței a două date calendaristice în ani, luni sau alte tipuri de date INTERVAL. Sprijinul vine însă de la o funcție proprietară - AGE care, după cum îi spune numele, calculează intervalul scurs de la o anumită dată până la momentul curent (atunci când funcția are un singur parametru) sau intervalul scurs între două date/momente (atunci când funcția are doi parametri. Mai mult, funcția AGE poate fi argument al funcției EXTRACT, așa încât libertatea de mișcare este aproape totală în materie de intervale. Pentru exemplificare, dorim să exprimăm intervalul scurs între data emiterii fiecăreia dintre facturi și 22 noiembrie 2007 în zile, în ani-luni-zile, numai în ani sau numai în luni:

```
SELECT NrFact, DATE'2008-11-22', DataFact,
        DATE'2008-11-22' - DataFact AS Zile_Scurse,
        AGE(DATE'2008-11-22',DataFact) AS Interval_Scurs,
        EXTRACT (YEAR FROM AGE(DATE'2008-11-22',DataFact)) AS Interval_Ani,
        EXTRACT (YEAR FROM AGE(DATE'2008-11-22',DataFact)) * 12 +
        EXTRACT (MONTH FROM AGE(DATE'2008-11-22',DataFact)) AS Interval_Luni
FROM facturi
```

NRFACT	DATE'2008-11-22'	DATAFACT	ZILE_SCURSE	ANILUNI_SC	LUNI_SC1	LUNI_SC2
1111	22-11-2008	01-08-2007	479	1-4	16	15,67
1112	22-11-2008	01-08-2007	479	1-4	16	15,67
1113	22-11-2008	01-08-2007	479	1-4	16	15,67
1114	22-11-2008	01-08-2007	479	1-4	16	15,67
1115	22-11-2008	02-08-2007	478	1-4	16	15,64
1116	22-11-2008	02-08-2007	478	1-4	16	15,64
1117	22-11-2008	03-08-2007	477	1-4	16	15,61
1118	22-11-2008	04-08-2007	476	1-4	16	15,58
1119	22-11-2008	07-08-2007	473	1-4	16	15,48
1120	22-11-2008	07-08-2007	473	1-4	16	15,48
1121	22-11-2008	07-08-2007	473	1-4	16	15,48
1122	22-11-2008	07-08-2007	473	1-4	16	15,48
2111	22-11-2008	14-08-2007	466	1-3	15	15,25
2112	22-11-2008	14-08-2007	466	1-3	15	15,25
2113	22-11-2008	14-08-2007	466	1-3	15	15,25
2115	22-11-2008	15-08-2007	465	1-3	15	15,22
2116	22-11-2008	15-08-2007	465	1-3	15	15,22

Figura 6.23. Funcții (Oracle) EXTRACT, TRUNC și MONTHS_BETWEEN

Rezultatul este cel din figura 6.24. Pentru calculul numărului de luni am înmulțit numărul anilor cu 12 la care am adăugat numărul lunilor extras cu EXTRACT (MONTH...).

nrfact numeric(8,0)	date date	datafact date	zile_scurse integer	interval_scurs interval	interval_ani double precis	interval_luni double precis
1111	2008-11-22	2007-08-01	479	1 year 3 mons 21 days	1	15
1112	2008-11-22	2007-08-01	479	1 year 3 mons 21 days	1	15
1113	2008-11-22	2007-08-01	479	1 year 3 mons 21 days	1	15
1114	2008-11-22	2007-08-01	479	1 year 3 mons 21 days	1	15
1115	2008-11-22	2007-08-02	478	1 year 3 mons 20 days	1	15
1116	2008-11-22	2007-08-02	478	1 year 3 mons 20 days	1	15
1117	2008-11-22	2007-08-03	477	1 year 3 mons 19 days	1	15
1118	2008-11-22	2007-08-04	476	1 year 3 mons 18 days	1	15
1119	2008-11-22	2007-08-07	473	1 year 3 mons 15 days	1	15
1120	2008-11-22	2007-08-07	473	1 year 3 mons 15 days	1	15
1121	2008-11-22	2007-08-07	473	1 year 3 mons 15 days	1	15
1122	2008-11-22	2007-08-07	473	1 year 3 mons 15 days	1	15
2111	2008-11-22	2007-08-14	466	1 year 3 mons 8 days	1	15
2112	2008-11-22	2007-08-14	466	1 year 3 mons 8 days	1	15

Figura 6.24. Funcţii (PostgreSQL) AGE şi EXTRACT

Şi SQL Server are tot o funcţie proprietară pentru calculul intervalului dintre două date calendaristice - DATEDIFF.

```
SELECT NrFact, '2007-11-22', DataFact,
        DATEDIFF(YEAR, DataFact, '2007-11-22') AS Ani_Scursi,
        DATEDIFF(YEAR, DataFact, '2007-11-22') * 12 +
                DATEDIFF(MONTH, DataFact, '2007-11-22') AS Luni_Scurse,
        DATEDIFF(DAY, DataFact, '2007-11-22') AS Zile_Scurse
FROM facturi
```

Rezultatul este cel din figura 6.25.

NrFact	(No column name)	DataFact	Ani_Scursi	Luni_Scurse	Zile_Scurse
1111	2007-11-22	2007-08-01 00:00:00	0	3	113
1112	2007-11-22	2007-08-01 00:00:00	0	3	113
1113	2007-11-22	2007-08-01 00:00:00	0	3	113
1114	2007-11-22	2007-08-01 00:00:00	0	3	113
1115	2007-11-22	2007-08-02 00:00:00	0	3	112
1116	2007-11-22	2007-08-02 00:00:00	0	3	112
1117	2007-11-22	2007-08-03 00:00:00	0	3	111
1118	2007-11-22	2007-08-04 00:00:00	0	3	110
1119	2007-11-22	2007-08-07 00:00:00	0	3	107
1120	2007-11-22	2007-08-07 00:00:00	0	3	107

Figura 6.25. Funcţia MS SQL Server DATEDIFF

În DB2, după cum am văzut în paragraful 5.3, la o dată calendaristică poate fi adunat un interval de tip an, lună, zi etc. fără a folosi explicit clauza INTERVAL. Tot acolo am văzut că diferenţa dintre două calendaristice este un număr care conţine, de la dreapta la stânga, numărul de zile, de luni şi de ani. Pentru calculul numărului de zile dintre două date se poate folosi o altă funcţie - DAYS. Această funcţie returnează numărul de zile corespunzător unei date calendaristice. Prin scăderea a două funcţii DAYS obţinem tocmai intervalul în zile dintre cele două date:

```
SELECT NrFact, '2007-11-22' AS Data_Reper, DataFact,
        DataFact - '2007-11-22' AS Ani_Luni_Zile_Scurse,
        DAYS('2007-11-22') - DAYS(DataFact) AS Zile_Scurse
FROM facturi
```

Ultimele două coloane ale rezultatului din figura 6.26 sunt edificatoare.

NRFACT	DATA_REPER	DATAFACT	ANI_LUNI_ZILE_SCURSE	ZILE_SCURSE
1.111	2007-11-22	01.08.2007	-321	113
1.112	2007-11-22	01.08.2007	-321	113
1.113	2007-11-22	01.08.2007	-321	113
1.114	2007-11-22	01.08.2007	-321	113
1.115	2007-11-22	02.08.2007	-320	112
1.116	2007-11-22	02.08.2007	-320	112
1.117	2007-11-22	03.08.2007	-319	111
1.118	2007-11-22	04.08.2007	-318	110
1.119	2007-11-22	07.08.2007	-315	107
1.120	2007-11-22	07.08.2007	-315	107
1.121	2007-11-22	07.08.2007	-315	107
1.122	2007-11-22	07.08.2007	-315	107
2.111	2007-11-22	14.08.2007	-308	100
2.112	2007-11-22	14.08.2007	-308	100

Figura 6.26. Două diferențe între două date calendaristice în DB2

Cea mai spumoasă funcție DB2 dedicată extragerii unor informații dintr-un interval cuprins între două momente este TIMESTAMPDIFF, care are două argumente, dintre care primul semnalizează tipul intervalului și poate avea valorile:

- 1 pentru fracțiuni de secundă;
- 2 pentru secunde;
- 4 pentru minute;
- 8 pentru ore;
- 16 pentru zile;
- 32 pentru săptămâni;
- 64 pentru luni;
- 128 pentru trimestre și
- 256 pentru ani,

iar al doilea este chiar diferența dintre cele două momente, ce trebuie transformată în șir de caractere (funcția CHAR):

```
SELECT TIMESTAMP('2008-03-27 14:12:55') AS Moment_Start,
        TIMESTAMP('2006-12-10 23:44:32') AS Moment_Stop,
        TIMESTAMPDIFF(16, CHAR(
        TIMESTAMP('2008-03-27 14:12:55') - TIMESTAMP('2006-12-10 23:44:32'))
                        ) AS Zile_Scurse,
        TIMESTAMPDIFF(32, CHAR(
        TIMESTAMP('2008-03-27 14:12:55') - TIMESTAMP('2006-12-10 23:44:32'))
                        ) AS Sapt_Scurse,
        TIMESTAMPDIFF(64, CHAR(
        TIMESTAMP('2008-03-27 14:12:55') - TIMESTAMP('2006-12-10 23:44:32'))
```

```
                    ) AS Luni_Scurse
FROM sysibm.sysdummy1
```

Diferenţa dintre cele două momente este exprimată, pe rând, în zile, săptămâni şi ani - vezi figura 6.27.

MOMENT_START	MOMENT_STOP	ZILE_SCURSE	SAPT_SCURSE	LUNI_SCURSE
27.03.2008 14:12:55 000000	10.12.2006 23:44:32 000000	471	67	15

Figura 6.27. Diferenţa (exprimată, pe rând, în zile, săptămâni şi luni) dintre două momente în DB2

6.5. Conversii între tipuri de date

Funcţia de conversie prezentă în mai toate limbajele de programare cunoscute este CAST pe care am întrebuinţat-o deja în paragraful 5.3 când precizam că în standardele SQL şi unele SGBD-uri (de exemplu, IBM DB2, SQL Server) concatenarea unor şiruri de caractere cu numere sau date calendaristice reclamă conversia numerelor/datelor şi intervalelor calendaristice în şiruri. Afişăm un text legat de scadenţa fiecărei facturi emise în luna septembrie 2007, text care să folosească funcţiile CAST, TRIM, ..., plus EXTRACT în clauza WHERE, precum şi o expresie dată calendaristică + interval.

```
SELECT 'Scadenta facturii ' || TRIM( TRAILING FROM CAST (NrFact AS CHAR(8))
        ) || ' (trimisa clientului ' || dencl ||
        ') are scadenta pe ' || CAST ((DataFact + INTERVAL '14' DAY) AS CHAR(10)
                        ) AS Scadente_facturi_Sept2007
FROM facturi f INNER JOIN clienti c ON f.CodCl=c.CodCl
WHERE EXTRACT (YEAR FROM datafact) = 2007 AND
        EXTRACT (MONTH FROM datafact) = 9
```

A sosit momentul să recunoaştem că fără funcţia CAST (sau CONVERT) în MS SQL Server nu putem obţine rezultatul din figura 6.19. Fiecare SELECT „reunit" din interogarea următoare funcţionează izolat:

```
SELECT 'Momentul:' AS Parametru, '2007-11-10 23:55:10' AS Valoare
        UNION
SELECT 'Zi din sapt.:', DATEPART (WEEKDAY, '2007-11-10 23:55:10')
        UNION
SELECT 'Zi din an:', DATEPART (DAYOFYEAR, '2007-11-10 23:55:10')
        UNION
SELECT 'Sapt. din an:', DATEPART (WEEK, '2007-11-10 23:55:10')
        UNION
SELECT 'Trim. din an:', DATEPART (QUARTER, '2007-11-10 23:55:10')
```

În schimb când lansăm întreaga interogare de mai sus obţinem un mesaj de eroare (vezi figura 6.28) din cauza faptului că trucul concatenării cu un spaţiu nu transformă automat a doua coloană într-un şir de caractere.

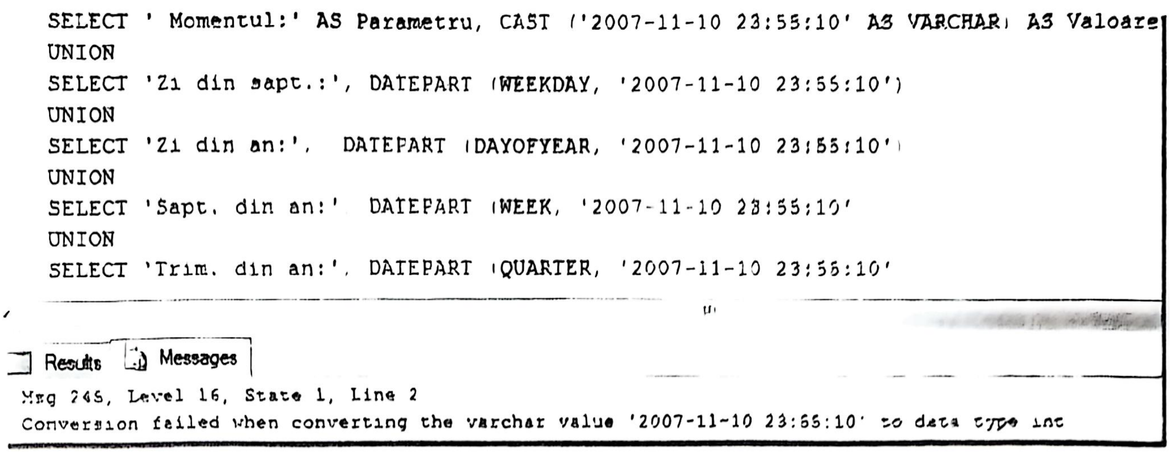


Figura 6.28. O mică problemă la reuniune în MS SQL Server

Cu ajutorul funcției CAST, lucrurile se rezolvă destul de elegant:

```
SELECT 'Momentul:' AS Parametru,
        CAST ('2007-11-10 23:55:10' AS VARCHAR) AS Valoare
        UNION
SELECT 'Zi din sapt.:',
        CAST (DATEPART (WEEKDAY, '2007-11-10 23:55:10') AS VARCHAR)
        UNION
SELECT 'Zi din an:',
        CAST (DATEPART (DAYOFYEAR, '2007-11-10 23:55:10')  AS VARCHAR)
        UNION
SELECT 'Sapt. din an:', CAST (DATEPART (WEEK, '2007-11-10 23:55:10') AS VARCHAR)
        UNION
SELECT 'Trim. din an:',
        CAST (DATEPART (QUARTER, '2007-11-10 23:55:10') AS VARCHAR)
```

Dacă ar fi să rescriem interogarea de mai sus cu, exclusiv, funcțiile proprietare SQL Server, am putea apela la: CONVERT (pe post de CAST) şi STR (pentru conversia unui număr în şir de caractere). Funcțiile DATEPART, DAY, MONTH şi YEAR returnează numere. Pentru afişarea numelui unei componente dintr-o dată calendaristică, în loc de DATEPART se întrebuințează DATENAME. Noua variantă (cu adăugiri) a interogării (cu rezultatul în figura 6.29) este:

```
SELECT 'Momentul:' AS Parametru,
        CONVERT (VARCHAR, '2007-11-10 23:55:10') AS Valoare
        UNION
SELECT 'Zi din sapt.:', STR (DATEPART (WEEKDAY, '2007-11-10 23:55:10'))
        UNION
SELECT 'Zi din an:', STR (DATEPART (DAYOFYEAR, '2007-11-10 23:55:10'))
        UNION
SELECT 'Sapt. din an:', STR (DATEPART (WEEK, '2007-11-10 23:55:10') )
        UNION
SELECT 'Trim. din an:', STR (DATEPART (QUARTER, '2007-11-10 23:55:10'))
        UNION
SELECT 'Numele zilei:', DATENAME (WEEKDAY, '2007-11-10 23:55:10')
```

Parametru	Valoare
Momentul:	2007-11-10 23.55.10
Numele zilei:	Saturday
Sapt. din an:	45
Trim din an:	4
Zi din an:	314
Zi din sapt.:	7

Figura 6.29. Câteva funcţii de conversie a componenteleor unei date calendaristice în SQL Server

Una dintre cele mai puternice funcţii de conversie din Oracle şi PostgreSQL este TO_CHAR, la care se adaugă TO_DATE şi TO_NUMBER. Spun puternică, deoarece, cunoscând tipologia caracterelor folosite în şablonul acestor funcţii, pot fi obţinute informaţii diverse. Să luăm câteva exemple. Înainte de implementarea opţiunii DATE'YYYY-MM-DD', şi în Oracle şi în PostgreSQL cel mai sigur mod de specificare a unui literal de tip dată calendaristică presupunea folosirea funcţiei TO_DATE. Clauza WHERE din comanda SELECT anterioară se putea scrie:

```
SELECT ...
FROM ...
WHERE DataFact BETWEEN TO_DATE('01/09/2007', 'DD/MM/YYYY')
        AND TO_DATE('30/09/2007', 'DD/MM/YYYY')
```

Ordinea componentelor unei date calendaristice este indicată de şablon, utilizatorul având o mare libertate de mişcare. Preţul plătit este lungimea, şablonul fiind considerabil mai lung prin comparaţie cu *DATE'yyyy-mm-dd'*. Cele mai de efect opţiuni pot fi gustate, însă, la operaţiunea inversă, de conversie din dată calendaristică în şir de caractere – TO_CHAR – conversie care coincide cu extragerea unor informaţii utile din data/momentul argument. Astfel, similar figurii 6.17 (paragraful 6.3) se poate obţine rezultatul (Oracle) din figura 6.30 a interogării următoare:

```
SELECT DATE'2007-09-17',
        TO_CHAR(DATE'2007-09-17', 'yyyy') AS anul,
        TO_CHAR(DATE'2007-09-17', 'mm') AS luna,
        TO_CHAR(DATE'2007-09-17', 'dd') AS ziua
FROM dual
```

DATE'2007-09-17'	ANUL	LUNA	ZIUA
17-09-2007	2007	09	17

Figura 6.30. Test: identificaţi trei diferenţe faţă de figura 6.17

Cei cu dioptrii foarte mari vor remarca faptul că anul, luna şi ziua sunt aliniate la stânga în figura 6.30, în timp ce în figura 6.17 erau aliniate la dreapta. Aceasta deoarece rezultatul unei funcţii TO_CHAR este un şir de caractere, deci va fi aliniat la stânga, în timp ce rezultatele funcţiei EXTRACT erau numere. Pentru a „transforma" anul, luna şi ziua în numere, încadrăm funcţia TO_CHAR într-o funcţie TO_NUMBER:

```
SELECT DATE'2007-09-17',
        TO_NUMBER(TO_CHAR(DATE'2007-09-17', 'yyyy')) AS anul,
        TO_NUMBER(TO_CHAR(DATE'2007-09-17', 'mm')) AS luna,
        TO_NUMBER(TO_CHAR(DATE'2007-09-17', 'dd')) AS ziua
FROM dual
```

Atunci când, la conversia în număr, dorim un anumit număr de cifre pe care să se facă reprezentarea, funcția TO_NUMBER poate fi folosită împreună cu un șablon în care să se indice numărul pozițiilor pentru partea întreagă și pentru partea fracționară.

Haideți să luăm în discuție o problemă „resimțită" încă de la prima ediție a cărții și actuală în *PostgreSQL pgAdmin* și *MS SQL Server Management Studio* la afișarea rezultatelor în interogări în care intervin calcule aritmetice (de exemplu, pentru determinarea valorii unei linii dintr-o factură sau a unei facturi, sau la calculul TVA). Modul în care se prezintă coloanele calculate nu e deloc entuziasmant. Este drept, e o problemă de afișare, deci de interfață/utilitar de conectare la bază, nu de SQL! Să luăm un exemplu în care trebuie să determinăm, pentru fiecare linie a facturii cu numărul 1111, valoarea fără TVA și TVA-ul liniei. Aceasta presupune joncționarea LINIIFACT cu PRODUSE, întrucât procentul care se aplică la vânzarea unui produs se află în tabela PRODUSE (atributul ProcTVA). Interogarea:

```
SELECT lf.*, Cantitate * PretUnit AS ValFaraTVA,
        Cantitate * PretUnit * ProcTVA AS TVA_linie
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
```

executată în Oracle SQL*Plus (sau SQL Developer) obține un rezultat care doar cu indulgență poate fi catalogat drept tabelar (figura 6.31); aceasta din pricina poziției fracționare diferite de la o linie la alta, ceea ce strică alinierea coloanei TVA_Linie.

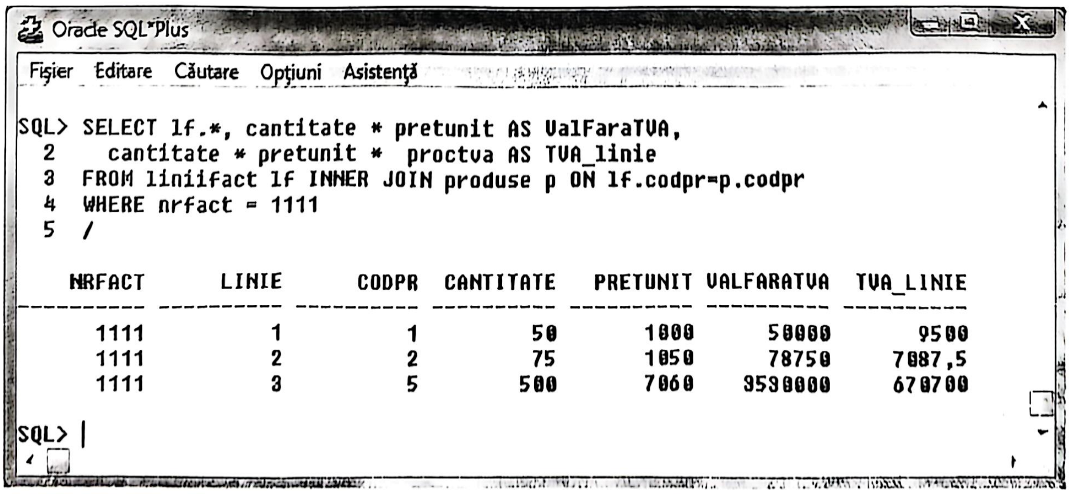


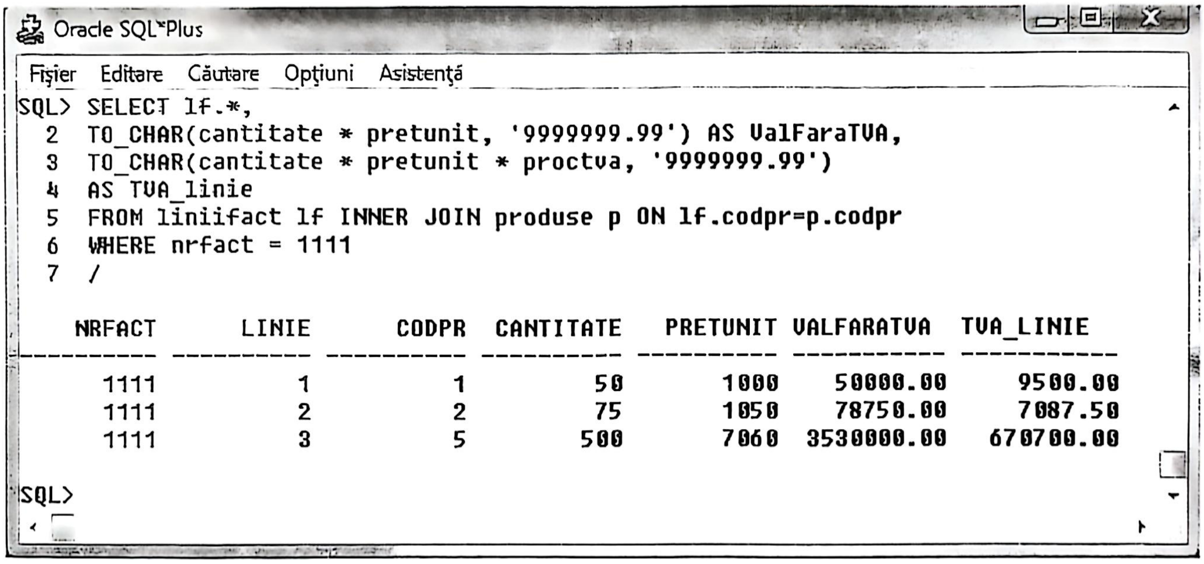
Figura 6.31. Modul de afișare a coloanelor numerice în Oracle SQL*Plus

Ei bine, pentru a mai salva din stilul de afișare (și, implicit, aparențele) am folosit în multe figuri[9] funcția TO_CHAR, în genul:

9. În prima ediție a cărții, interogările Oracle au fost executate în SQL*Plus.

SELECT lf.*,
TO_CHAR(Cantitate * PretUnit, '9999999.99') AS ValFaraTVA,
TO_CHAR(Cantitate * PretUnit * ProcTVA, '9999999.99') AS TVA_linie
FROM liniifact lf INNER JOIN produse p ON lf. CodPr =p. CodPr
WHERE NrFact = 1111

Precizând şapte cifre la partea întreagă (de şapte ori cifra 9) şi două poziţii fracţionare (cei doi de 9 care urmează punctului), am forţat, odată cu conversia, alinierea sub formă tabelară - vezi figura 6.32. Din păcate, artificiul funcţionează numai în Oracle SQL*Plus, pentru că interfaţa este în mod text, fiecare caracter având aceeaşi lungime, în timp ce interfaţa Oracle SQL Developer este grafică, spaţiul rezervat reprezentării unui caracter fiind diferit, în funcţie de formă (un I este mult mai îngust decât un M).



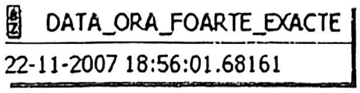

```
SQL> SELECT lf.*,
  2  TO_CHAR(cantitate * pretunit, '9999999.99') AS ValFaraTVA,
  3  TO_CHAR(cantitate * pretunit * proctva, '9999999.99')
  4  AS TVA_linie
  5  FROM liniifact lf INNER JOIN produse p ON lf.codpr=p.codpr
  6  WHERE nrfact = 1111
  7  /

    NRFACT      LINIE      CODPR  CANTITATE   PRETUNIT VALFARATVA  TVA_LINIE
---------- ---------- ---------- ---------- ---------- ----------- -----------
      1111          1          1         50       1000    50000.00     9500.00
      1111          2          2         75       1050    78750.00     7087.50
      1111          3          5        500       7060  3530000.00   670700.00

SQL>
```

Figura 6.32. Ameliorarea modului de afişare a coloanelor numerice în Oracle SQL*Plus cu ajutorul funcţiei TO_CHAR

Dacă în şabloanele funcţiilor TO_DATE/TO_CHAR dedicate datelor calendaristice foloseam DD pentru zile, MM pentru luni şi YYYY pentru an, pentru ore se poate folosi fie HH, fie HH24 (HH afişează orele pe formatul 0-12AM, apoi 0-12PM, în timp ce HH24 permite formatul 0-24), pentru minute MI, iar pentru secunde SS. Spre exemplu, următoarea interogare a fost pregătită pentru lansarea în execuţie pe 22 noiembrie 2007 pe la 18:55:

SELECT TO_CHAR(CURRENT_TIMESTAMP,
'DD-MM-YYYY HH24:MI:SS.SSSSS') AS Data_Ora_Foarte_Exacte
FROM dual

Până la lansarea în execuţie a interogării a trecut aproape un minut, aşa că rezultatul este cel din figura 6.33.



DATA_ORA_FOARTE_EXACTE
22-11-2007 18:56:01.68161

Figura 6.33. Afişarea tuturor componentelor datei/orei curente (la ora execuţiei)

După cum știți, de mai bine de zece ani a fost introdus în România sistemul de identificare al persoanelor bazat pe codul numeric personal (CNP-ul). CNP-ul este o „cheie inteligentă", în sensul că înglobează câteva informații despre „posesor":

- prima cifră indică sexul: dacă este 1 sau 5 este vorba despre un băiat/bărbat; dacă este 2 sau 6 este o fată/femeie (mai sunt și alte situații, dar nu le tratăm);
- următoarele șase cifre indică data nașterii (două pentru an, două pentru lună și două pentru zi; dacă persoana este născută înainte de 2000, atunci „cifra sexului" este 1 sau 2; pentru cei născuți după 2000, „cifra sexului" este 5 sau 6 etc.

Pentru a afla data nașterii pacienților născuți înainte de anul 2000, formulăm o interogare din care extragem cifrele de la 2 la 7 inclusiv. După cum se observă în figura 6.34, la extragere, secolul implicit este cel actual, astfel încât persoanele par născute în viitor.

```
SELECT NumePacient, cnp,
        TO_DATE(SUBSTR(cnp, 2, 6), 'YYMMDD') AS Data_Nasterii_CNP
FROM pacienti
WHERE SUBSTR(cnp, 1, 1) IN ('1','2')
```

NUMEPACIENT	CNP	DATA_NASTERII_CNP
Stroe Mihaela	2601228390834	28-12-2060
Buzatu Corneliu	1650512370514	12-05-2065
Bagdasar Adela	2601002250611	02-10-2060
Popescu Maria-Mirabela	2721231300888	31-12-2072
Cazan Ana Maria	2690202200345	02-02-2069

Figura 6.34. Tentativă de extragere a datei din CNP pentru cei născuți în secolul trecut

Soluția pare simplă: scădem 100 de ani din data obținută mai sus:

```
SELECT NumePacient, cnp,
        TO_DATE(SUBSTR(cnp, 2, 6), 'YYMMDD') -
                        INTERVAL '100' YEAR AS Data_Nasterii_CNP
FROM pacienti
WHERE SUBSTR(cnp, 1, 1) IN ('1','2')
```

Paradoxal, însă, Oracle tratează această variantă cu oarecare răceală, dacă este să ne luăm după mesajul din figura 6.35.

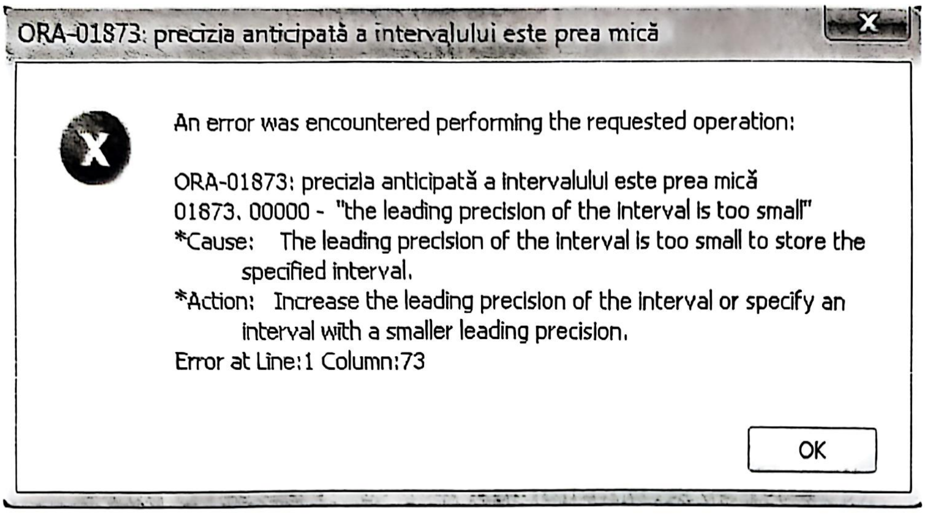


Figura 6.35. Surprize, surpize... în Oracle (episodul 2)

Nedumerirea este cu atât mai mare cu cât, dacă în locul INTERVAL, folosim funcţia „proprietară" Oracle ADD_MONTHS cu argumentul -1200 (de luni), lucrurile sunt în regulă – vezi figura 6.36:

```
SELECT NumePacient, cnp,
        ADD_MONTHS(TO_DATE(SUBSTR(cnp, 2, 6), 'YYMMDD'),
                                -1200)
        AS Data_Nasterii_CNP
FROM pacienti
WHERE SUBSTR(cnp, 1, 1) IN ('1','2')
```

NUMEPACIENT	CNP	DATA_NASTERII_CNP
Stroe Mihaela	2601228390834	28-12-1960
Buzatu Corneliu	1650512370514	12-05-1965
Bagdasar Adela	2601002250611	02-10-1960
Popescu Maria-Mirabela	2721231300868	31-12-1972
Cazan Ana Maria	2690202200345	02-02-1969

Figura 6.36. Extragerea corectă a datei din CNP pentru cei născuţi în secolul trecut

Iată o interogare în care folosim şi alte formate de extragere şi afişare a lunii dintr-o dată calendaristică:

```
SELECT NrFact, DataFact,
        TO_CHAR(DataFact, 'MM') AS Luna_Nr,
        TO_CHAR(DataFact, 'MONTH') AS Luna_Sir_Intreaga_Majuscule,
        TO_CHAR(DataFact, 'month') AS Luna_Sir_Intreaga_Minuscule,
        TO_CHAR(DataFact, 'MON') AS Luna_Sir_Prescurtat
FROM facturi
WHERE NrFact = 1111
```

Clauzele *MONTH* şi *month* determină afişarea (vezi figura 6.37) numelui întreg al lunii (în engleză sau română, depinde de modul în care a fost instalat sistemul de operare, serverul Oracle şi SQL Developer), în timp ce *MON* (sau *mon*) afişează numele prescurtat, din trei litere, al lunii.

NRFACT	DATAFACT	LUNA_NR	LUNA_SIR_INTREAGA_MAJUSCULE	LUNA_SIR_INTREAGA_MINUSCULE	LUNA_SIR_PRESCURTAT
1111	01-08-2007	08	AUGUST	august	AUG

Figura 6.37. Modalități de vizualizare a unei luni dintr-o dată calendaristică

Următoarea interogare afişează, oarecum în premieră, numele zilei din săptămână în care cade o anumită dată calendaristică, folosindu-se drept şablon *DAY* - vezi figura 6.38:

```
SELECT NrFact, DataFact,
        TO_CHAR(DataFact, 'DD') AS Zi_Din_Luna,
        TO_CHAR(DataFact, 'DAY') AS Zi_Sapt_char
FROM facturi
WHERE TO_CHAR(DataFact, 'MM-YYYY') = '09-2007'
```

În plus, am simplificat mult şi predicatul de extragere numai a facturilor din luna septembrie 2007.

NRFACT	DATAFACT	ZI_DIN_LUNA	ZI_SAPT_CHAR
3111	01-09-2007	01	SÂMBĂTĂ
3112	01-09-2007	01	SÂMBĂTĂ
3113	02-09-2007	02	DUMINICĂ
3115	02-09-2007	02	DUMINICĂ
3116	10-09-2007	10	LUNI
3117	10-09-2007	10	LUNI
3118	17-09-2007	17	LUNI

Figura 6.38. Vizualizarea zilei din săptămână

După cum spuneam în paragraful 6.3, PostgreSQL-ul are un argument pentru funcția EXTRACT care permite aflarea zilei din săptămână, dar numai sub formă numerică (de la 0 pentru duminică până la 6 pentru sâmbătă). Pentru ilustrare, adăugăm în clauza SELECT a interogării anterioare o coloană în care folosim această opțiune (rezultatul este în figura 6.39):

```
SELECT NrFact, DataFact,
        TO_CHAR(DataFact, 'DD') AS Zi_Din_Luna,
        TO_CHAR(DataFact, 'DAY') AS Zi_Sapt_char,
        EXTRACT (DOW FROM DataFact) AS Zi_Sapt_num
FROM facturi
WHERE TO_CHAR(DataFact, 'MM-YYYY') = '09-2007'
```

nrfact numeric(8,0)	datafact date	zi_din_luna text	zi_sapt_char text	zi_sapt_num double precis
3111	2007-09-01	01	SATURDAY	6
3112	2007-09-01	01	SATURDAY	6
3113	2007-09-02	02	SUNDAY	0
3115	2007-09-02	02	SUNDAY	0
3116	2007-09-10	10	MONDAY	1
3117	2007-09-10	10	MONDAY	1
3118	2007-09-17	17	MONDAY	1

Figura 6.39. Funcția EXTRACT (DOW...) în PostgreSQL

Atenţie la predicatele în care se compară ziua săptămânii obţinută cu ajutorul funcţiei TO_CHAR! Dacă am dori să afişăm numai facturile întocmite lunea în luna septembrie 2007, am fi tentaţi să redactăm SELECT-ul astfel:

```
SELECT NrFact, DataFact,
        TO_CHAR(DataFact, 'DD') AS Zi_Din_Luna,
        TO_CHAR(DataFact, 'DAY') AS Zi_Sapt_char
FROM facturi
WHERE TO_CHAR(DataFact, 'MM-YYYY') = '09-2007'
        AND TO_CHAR(DataFact, 'DAY') IN ('LUNI', 'MONDAY')
```

Ei bine, interogarea nu extrage niciodată nimic, iar răspunsul e frustrant: TO_CHAR converteşte ziua într-un şir de caractere de lungime fixă. Cum nu prea avem noroc să nimerim din prima lungimea de comparaţie, cel mai sănătos este să folosim o funcţie care elimină spaţiile de la final - TRIM (TRAILING...).

```
SELECT NrFact, DataFact,
        TO_CHAR(DataFact, 'DD') AS Zi_Din_Luna,
        TO_CHAR(DataFact, 'DAY') AS Zi_Sapt_char
FROM facturi
WHERE TO_CHAR(DataFact, 'MM-YYYY') = '09-2007' AND
      TRIM (TRAILING FROM TO_CHAR(DataFact, 'DAY')) IN ('LUNI', 'MONDAY')
```

În DB2, după cum am văzut în paragrafele anterioare, obţinerea oricărei informaţii dintr-o dată calendaristică sau moment presupune folosirea unei funcţii speciale: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MICROSECOND, DAYOFWEEK etc. De asemenea, exprimarea unui interval precizat prin diferenţa dintre două date/momente se realizează prin funcţia TIMESTAMPDIFF. Pentru conversia dintr-un tip în altul, funcţia CAST este, în general, suficientă. Pe lângă acestea, mai pot fi puse la lucru şi câteva alte funcţii (mai mult sau mai puţin) „proprietare", dintre care amintim:

- CHAR converteşte un număr, dată sau moment în şir de caractere[10];
- DATE converteşte un şir de caractere într-o dată calendaristică sau extrage data dintr-un moment;
- INTEGER converteşte un şir de caractere, dată calendaristică sau moment într-un număr întreg;
- TIME converteşte un şir de caractere într-o „oră" sau extrage ora-minutul-secunda-fracţiunea de secundă dintr-un moment;
- TIMESTAMP converteşte o expresie într-un moment.

Iată o interogare ce foloseşte câteva dintre aceste funcţii şi rezultatul său (figura 6.40):

```
SELECT CHAR(CURRENT_TIMESTAMP) AS "CHAR(CURRENT_TIMESTAMP)",
       DATE(CURRENT_TIMESTAMP) AS "DATE(CURRENT_TIMESTAMP)",
       INTEGER(DATE(CURRENT_TIMESTAMP)) AS
              "INTEGER(DATE(CURRENT_TIMESTAMP))",
       TIMESTAMP(DATE('2008-03-23'), TIME('23:59:58')) AS "Un_TIMESTAMP"
```

10. Am folosit deja această funcţie la prezentarea TIMESTAMPDIFF.

FROM sysibm.sysdummy1

CHAR(CURRENT_TIMESTAMP)	DATE(CURRENT_TIMESTAMP)	INTEGER(DATE(CURRENT_TIMESTAMP))	Un_TIMESTAMP
2008-03-27-08.56.54.937000	27.03.2008	20080327	23.03.2008 23:59:58.000000

Figura 6.40. Funcții CHAR, DATE, INTEGER și TIMESTAMP în DB2

6.6. Alte funcții-sistem

Funcția CURRENT_DATE, pe care am mai folosit-o de câteva ori, poate fi încadrată și la categoria funcții-sistem. Returnează, în mai toate SGBD-urile, data curentă pe care o „știe" SGBD-ul și are numeroase rude prin dialectele SQL: SYSDATE[11] - Oracle, DATE() - FoxPro, GETDATE() - MS SQL Server etc.

CURRENT_TIME se folosește pentru aflarea orei (mai mult sau mai puțin exacte) curente, plus fusul orar, fiind implementată în Oracle și PostgreSQL. Fără zona ce indică fusul orar (GMT+2 pentru ora Bucureștiului), funcția devine LOCALTIME (PostgreSQL) - vezi figura 6.41.

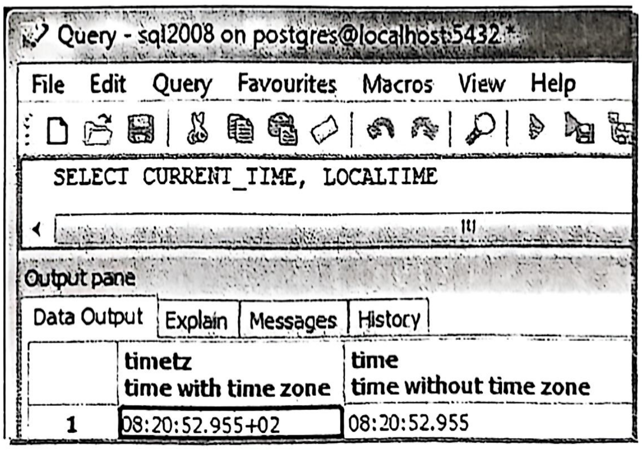


Figura 6.41. Diferența dintre CURRENT_TIME și LOCALTIME (PostgreSQL)

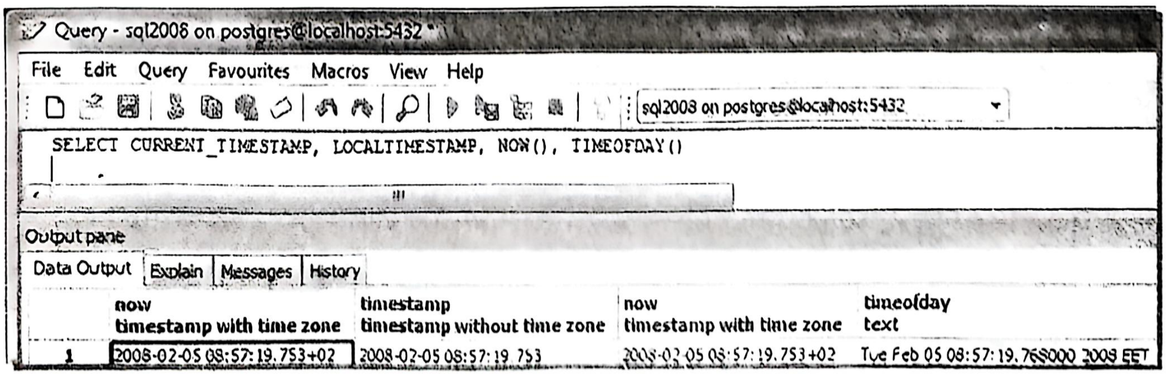


Figura 6.42. Momente curente (în PostgreSQL)

Diferența dintre „general" și local se menține și în cazul funcțiilor care furnizează momentul curent, CURRENT_TIMESTAMP și LOCAL_TIMESTAMP. **În PostgreSQL,** echivalentul funcției CURRENT_TIMESTAMP este NOW() și, în

---

11. **Nu uitați că tipul DATE** în Oracle este, de fapt, un TIMESTAMP.

plus, mai dispunem şi de funcţia TIMEOFDAY() ce returnează momentul curent ca un şir de caractere (vezi figura 6.42).

În MS SQL Server, ora locală şi cea „universală" (de exemplu, GMT) se obţin cu funcţiile GETDATE şi GETUTCDATE (vezi figura 6.43).

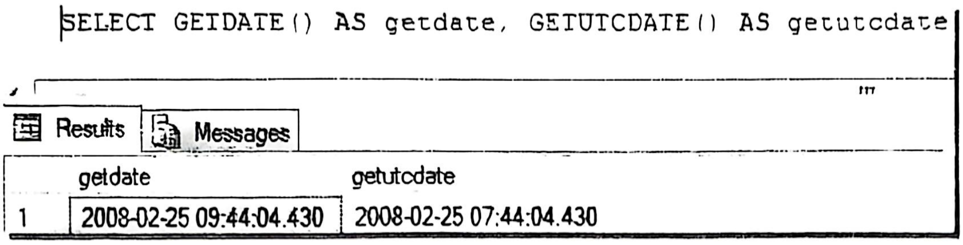


Figura 6.43. Momente curente (în MS SQL Server)

Toate funcţiile din acest paragraf puteau fi tratate în secţiunile dedicate datelor şi intervalelor. Următoarele, deşi tot „curente" (CURRENT_...) oferă informaţii necesare în multe aplicaţii, mai ales în cele client-server şi web:

- CURRENT_DATABASE (PostgreSQL) şi DB_NAME (MS SQL Server) furnizează numele bazei de date curente;
- CURRENT_SCHEMA (PostgreSQL), SCHEMA_NAME() (SQL Server) sau CURRENT SCHEMA (DB2) furnizează numele schemei în care se lucrează;
- CURRENT_USER (PostgreSQL, MS SQL Server), USER (Oracle, MS SQL Server, DB2, PostgreSQL), SESSION_USER (PostgreSQL, MS SQL Server) furnizează numele utilizatorului curent conectat la baza de date.

Modul lor de apelare, parametrii şi, uneori, chiar funcţionalitatea, pot diferi uşor de la un SGBD la altul. Figura 6.44 ilustrează modul de execuţie al interogării în PostgreSQL:

```
SELECT CURRENT_DATABASE(), CURRENT_SCHEMA(),
       CURRENT_USER, USER, SESSION_USER
```

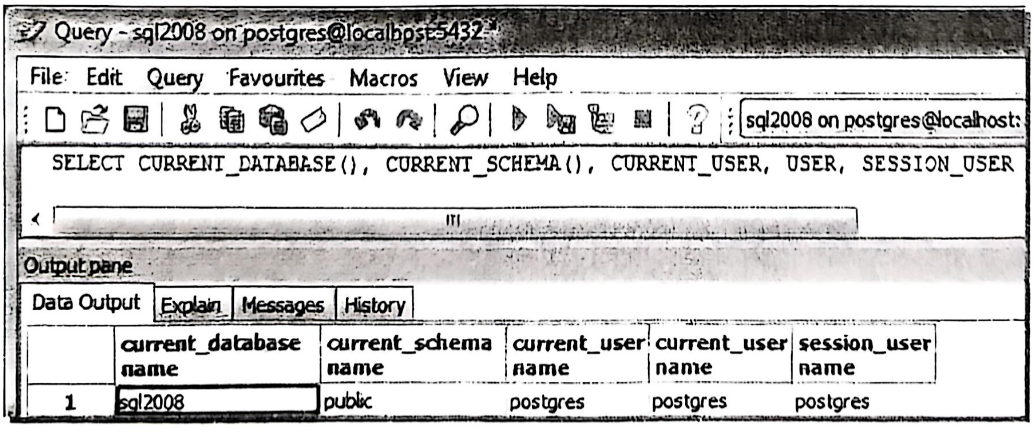


Figura 6.44. Baza de date, schema şi utilizatorul, curente, în PostgreSQL

6.7. Funcții-agregat

Denumirea acestui tip de funcții sugerează faptul că acestea se aplică unui set de valori şi returnează o singură valoare (de cele mai multe ori numerică). Cele mai folosite sunt: COUNT, SUM, MIN şi MAX. Formatul general al unei fraze SELECT ce conține funcții-agregat este:

SELECT funcţia-predefinită1, ... , funcţia-predefinităN
FROM listă-tabele
WHERE condiţii.

În lipsa opțiunii GROUP BY (tratată în capitolul următor), dacă în clauza SELECT este prezentă o funcție-agregat, rezultatul va conține o singură linie.

6.7.1. COUNT

Funcția COUNT contorizează valorile nenule ale unei coloane sau numărul de linii dintr-un rezultat al unei interogări, altfel spus, în rezultatul unei consultări, COUNT numără câte valori diferite de NULL are o coloană specificată sau câte linii sunt.

Câți clienți are firma?

SELECT COUNT (*) AS NrClienti
FROM clienti

Prezența asteriscului ca argument al funcției COUNT are ca efect numărarea liniilor tabelei CLIENȚI. Rezultatul este prezentat în figura 6.45.

Figura 6.45. Numărul clienților

Tabela CLIENȚI are cheie primară atributul CodCl care nu poate avea valori nule; de aceea, este corectă şi soluția:

SELECT COUNT (CodCl) AS NrClienti
FROM clienti

Folosind concatenarea, se poate obține un rezultat de forma unui mesaj (vezi figura 6.46).

SELECT 'Firma are ' | | COUNT (*) | | ' clienti' AS Rezultat
FROM clienti

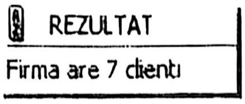


Figura 6.46. Altă formă de prezentare (în Oracle) a rezultatului funcției COUNT

Reamintim că, în DB2 sau SQL Server, pe lângă operatorul de concatenare (| | sau +) sau funcția CONCAT, este necesară conversia rezultatului funcției COUNT care este INTEGER în şir de caractere (CHAR), aşa că în DB2 putem scrie:

```
SELECT CONCAT ('Firma are ', CONCAT (
        CAST (COUNT (*) AS VARCHAR(5)), ' clienti')
                                        ) AS Rezultat
FROM clienti
```

iar în MS SQL Server:

```
SELECT 'Firma are ' + CAST (COUNT (*) AS VARCHAR(5)) + ' clienti'
        AS Rezultat
FROM clienti
```

Câte linii are produsul cartezian al tabelelor FACTURI şi LINIIFACT?

Suntem la rubrica „Răspundem ascultătorilor" (însă nu la Radio Erevan).

```
SELECT COUNT(*)
FROM facturi, liniifact
```

sau

```
SELECT COUNT(*)
FROM facturi CROSS JOIN liniifact
```

Oricare din cele două interogări returnează 1680 (în condiţiile în care liniile sunt exact cele din scriptul de inserare din capitolul 3.

Pentru câţi clienţi se cunoaşte adresa?

```
SELECT COUNT (Adresa) AS NrClienti_cu_adresa_cunoscuta
FROM clienti
```

Rezultatul, 5 (figura 6.47), putea fi obţinut şi ceva mai complicat, folosind în clauza *WHERE operatorul IS NULL* pe care-l tot amânăm pentru capitolul 8.

NrClienti_cu_adresa_cunoscuta
5

Figura 6.47. Numărul de linii din CLIENŢI cu adresă nenulă (SQL Server)

Câte facturi au fost emise pe 7 august 2007?

```
SELECT COUNT(NrFact) AS NrFacturi
FROM facturi
WHERE DataFact = DATE'2007-08-07'
```

Vă spun eu: patru.

Câte facturi au fost emise pentru clienţi din judeţul Vaslui?

Problema este că numărul facturilor poate fi determinat din tabela FACTURI, însă condiţia de selecţie, *Judet = 'Vaslui'*, este aplicabilă tabelei JUDEŢE. Întrucât cele două tabele nu pot fi joncţionate direct, se introduc în clauza FROM „intermediarii" CLIENŢI şi CODURI_POŞTALE:

```
SELECT COUNT(NrFact) AS NrFacturi
FROM facturi f
        INNER JOIN clienti c ON f.codcl = c.codcl
        INNER JOIN coduri_postale cp ON c.codpost = cp.codpost
```

INNER JOIN judete j ON cp.jud = j.jud
WHERE Judet = 'Vaslui'

Câte localități sunt preluate în baza de date?

Problema este un pic mai complicată, deoarece nu există o tabelă în care numele localității să fie cheie primară sau alternativă. Tabela CODURI_POŞTALE poate conține mai multe linii pentru aceleaşi oraşe, deoarece cele ceva mai mari prezintă mai multe coduri poştale.

Dacă ținem cont că o aceeaşi localitate (în mediul rural, mai ales) poate să apară în mai multe județe, interogarea pare a fi:

SELECT COUNT(Loc || Jud) AS NrLocalit
FROM coduri_postale[12]

NRLOCALIT_GRESIT
12

Figura 6.48. Folosirea eronată a funcției COUNT pentru aflarea numărului de localități

Necazul e că rezultatul obținut, 12 (vezi figura 6.48), este incorect, deoarece funcția COUNT numără toate valorile nenule ale atributului sau expresiei considerate ca argument. Din fericire, există o clauză prin care o valoare se ia în calcul o singură dată: DISTINCT. Rezultatul corect presupune varianta:

SELECT COUNT(DISTINCT Loc || Jud) AS NrLocalit
FROM coduri_postale

Valoarea furnizată de această interogare (vezi figura 6.49) este cea corectă, ținând cont de liniile introduse în tabelă în capitolul 3.

NRLOCALIT
8

Figura 6.49. Opțiunea DISTINCT a funcției COUNT (rezultat DB2)

6.7.2. SUM

Funcția SUM este una dintre cele mai utilizate funcții în aplicațiile economice, deoarece datele financiar-contabile şi cele ale evidenței tehnico-operative sunt preponderent cantitative. Probabil că prea multe explicații teoretice despre modul în care operează această funcție sunt inutile, aşa încât vom trece în revistă câteva exemple.

Care este valoarea fără TVA a facturii 1111 (figura 6.50)?

Valoarea unei facturi este dată de suma valorilor liniilor sale, fiecare linie descriind vânzarea unui produs sau serviciu. Cum valoarea fără TVA a fiecărei linii din factură este produsul *Cantitate * PrețUnit*, comanda SELECT are forma:

SELECT SUM(Cantitate * PretUnit) AS ValFaraTVA
FROM liniifact WHERE NrFact = 1111

12. Nu uitați că în MS SQL Server operatorul de concatenare nu e ||, ci +.

VALFARATVA_1111
3658750

Figura 6.50. Valoarea fără TVA a facturii 1111

Care este valoarea fără TVA a facturilor întocmite pe 7 august 2007?

Dacă valoarea fără TVA a facturilor se obţine din LINIIFACT, condiţia de selecţie se aplică asupra atributului DataFact din FACTURI, aşa că cele două tabele trebuie joncţionate, rezultatul fiind cel din figura 6.51.

```
SELECT SUM(Cantitate * PretUnit) AS ValFaraTVA_7aug2007
FROM liniifact lf INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE DataFact = DATE'2007-08-07'
```

VALFARATVA_7AUG2007
8941450

Figura 6.51. Vânzările fără TVA din 7 august 2007

Care sunt cele trei valori: fără TVA, TVA şi totală (cu TVA) ale facturii 1111?

Procentul de TVA care se aplică la vânzarea unui produs se află în tabela PRODUSE (atributul ProcTVA), aşa că în clauza FROM va fi adăugată pentru joncţiune şi această tabelă (valorile obţinute sunt cele din figura 6.52):

```
SELECT SUM(Cantitate * PretUnit) AS ValFaraTVA,
        SUM(Cantitate * PretUnit * ProcTVA) AS TVA,
        SUM(Cantitate * PretUnit + Cantitate * PretUnit * ProcTVA)
                AS ValTotala
FROM liniifact lf
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE f.NrFact = 1111
```

VALFARATVA	TVA	VALTOTALA
3,658,750.00	687,287.5000	4,346,037.5000

Figura 6.52. Cele trei valori ale facturii 1111

În condiţiile în care firma pentru care se foloseşte baza noastră de date comercializează exclusiv produse la care procentul TVA este unic - 19% - ar fi corectă şi varianta:

```
SELECT SUM(Cantitate * PretUnit) AS ValFaraTVA,
        SUM(Cantitate * PretUnit * .19) AS TVA,
        SUM(Cantitate * PretUnit + Cantitate * PretUnit * .19) AS ValTotala
FROM liniifact WHERE NrFact = 1111
```

În plus, nici nu ar mai fi nevoie de atributul ProcTVA în PRODUSE. Nu recomand renunţarea la acest atribut, întrucât, după un timp, procentele TVA ale unor produse se vor modifica.

Pentru a ne mai dezmorți mouse-ul, vom afișa cele trei valori pe verticală, ca în figura 6.53, prilej de ceva reuniuni. Iată versiunea Oracle/PostgreSQL[13]:

```
SELECT '1' AS " ",'Valoarea fara TVA' AS "Factura 1111",
        TRUNC(SUM(Cantitate * PretUnit)) AS Suma
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
UNION
SELECT '2','TVA', TRUNC(SUM(Cantitate * PretUnit * ProcTVA))
FROM liniifact lf
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE lf.NrFact = 1111
UNION
SELECT '3','Valoarea totala', TRUNC(SUM(Cantitate*PretUnit*(1 + ProcTVA)))
FROM liniifact lf
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE lf.NrFact = 1111
```

	Factura 1111	SUMA
1	Valoarea fara TVA	3658750
2	TVA	687287
3	Valoarea totala	4346037

Figura 6.53. Cele trei valori ale facturii 1111 – afișare pe verticală

Nu întotdeauna rezultatul reflectă ordinea firească de procesare a liniilor. Este motivul pentru care am introdus în interogare o coloană (fără nume, de fapt, cu „numele" spațiu), cu un prim rol decorativ și un al doilea legat de afișarea liniilor în ordinea care ne interesează. Dacă nu am fi fost mulțumiți de ordinea afișării, am fi adăugat interogării clauza *ORDER BY 1*.

13. În DB2, funcția TRUNC are obligatoriu un al doilea argument care semnifică la câte poziții fracționare să se facă trunchierea, iar în MS SQL Server varianta funcțională a acestei interogări este:

```
SELECT '1' AS " ",'Valoarea fara TVA' AS "Factura 1111",
        ROUND(SUM(Cantitate * PretUnit),0,1) AS Suma
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
UNION
SELECT '2','TVA', ROUND(SUM(Cantitate * PretUnit * ProcTVA),0,1)
FROM liniifact lf INNER JOIN facturi f ON lf.NrFact = f.NrFact INNER JOIN produse p
ON lf.CodPr = p.CodPr
WHERE lf.NrFact = 1111
UNION
SELECT '3','Valoarea totala', ROUND(SUM(Cantitate*PretUnit*(1 + ProcTVA)),0,1)
FROM liniifact lf INNER JOIN facturi f ON lf.NrFact = f.NrFact
INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE lf.NrFact = 1111
```

La cât se situează cifra vânzărilor în data de 7 august 2007?

Cifra de afaceri reprezintă, de fapt, valoarea totală a vânzărilor, deci valoarea cu tot cu TVA a facturilor emise pe 7 august 2007 (figura 6.54):

```
SELECT '7 aug. 2007' AS Data,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
FROM liniifact lf
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DataFact = DATE'2007-08-07'
```

DATA	VINZARI
7 aug. 2007	10610000,5

Figura 6.54. Vânzările cu TVA din 7 august 2007

Care este valoarea vânzărilor pentru „Client 1 SRL" (figura 6.55)?

```
SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Client1
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
WHERE DenCl = 'Client 1 SRL'
```

VINZARI_CLIENT1
15061538

Figura 6.55. Vânzările către Client 1 SRL

Să se afişeze valoarile (fără TVA, TVA, cu TVA) liniilor facturii 1111, plus o linie de total.

Practic, suntem interesaţi de un raport asemănător celui din figura 6.56. Raportul are două tipuri de rânduri, cel corespunzător liniilor din tabela LINIIFACT şi cel de totalizare.

Linia	Produs	Cantitate	Pret Unitar	Val. fara TVA	TVA linie	Val cu TVA linie
1	Produs 1	50	1000	50000	9500	59500
2	Produs 2	75	1050	78750	7087	85837
3	Produs 5	500	7060	3530000	670700	4200700
101	TOTAL factura	(null)	(null)	3658750	687287	4346037

Figura 6.56. Un rând de total pe lângă liniile facturii 1111

Obţinerea lor presupune două SELECT-uri conectate prin UNION. Pentru ca reuniunea să funcţioneze, rezultatele celor două SELECT-uri trebuie să fie unicompatibile, adică să prezintă acelaşi număr de coloane, iar prima coloană-rezultat al primului SELECT să fie de acelaşi tip cu prima coloană-rezultat al celui de-ai doilea SELECT ş.a.m.d.

```
SELECT linie AS "Linia", DenPr AS "Produs", Cantitate AS "Cantitate",
        TRUNC(PretUnit) AS "Pret Unitar",
        TRUNC(Cantitate * PretUnit) AS "Val. fara TVA",
        TRUNC(Cantitate * PretUnit * ProcTVA) AS "TVA linie",
        TRUNC(Cantitate * PretUnit * (1 + ProcTVA)) AS "Val cu TVA linie"
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
        UNION
SELECT 101, ' TOTAL factura ',
        NULL,
        NULL,
        TRUNC(SUM(Cantitate * PretUnit)),
        TRUNC(SUM(Cantitate * PretUnit * ProcTVA)),
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)))
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
ORDER BY 1
```

Cele două valori NULL din SELECT-ul corespunzător rândului de total au fost necesare pentru că nu se pot totaliza cantitățile și prețurile unitare, atâta vreme cât produsele sunt diferite. Clauza ORDER BY forțează ordonarea după prima coloană. Atenție, nici în Oracle, nici în PostgreSQL nu funcționează în interogarea de mai sus sintaxa *ORDER BY linie*[14]!. În schimb, funcționează în MS SQL Server, dar aici trebuie să înlocuim TRUNC cu ROUND cu un parametru suplimentar:

```
SELECT linie AS "Linia", DenPr AS "Produs", Cantitate AS "Cantitate",
        ROUND(PretUnit,0,1) AS "Pret Unitar",
        ROUND(Cantitate * PretUnit,0,1) AS "Val. fara TVA",
        ROUND(Cantitate * PretUnit * ProcTVA,0,1) AS "TVA linie",
        ROUND(Cantitate * PretUnit * (1 + ProcTVA),0,1) AS "Val cu TVA linie"
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
        UNION
SELECT 101, ' TOTAL factura ', NULL,
        NULL,
        ROUND(SUM(Cantitate * PretUnit),0,1),
        ROUND(SUM(Cantitate * PretUnit * ProcTVA),0,1),
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0,1)
FROM liniifact lf INNER JOIN produse p ON lf.CodPr=p.CodPr
WHERE NrFact = 1111
ORDER BY linie
```

Dacă în loc de NULL am vrea să apară spațiu, lucrurile ar degenera, deoarece spațiul este de tip șir de caractere, iar cantitățile și prețurile unitare sunt numerice (vezi figura 6.57).

14. Ar funcționa, totuși, ORDER BY "Linia".

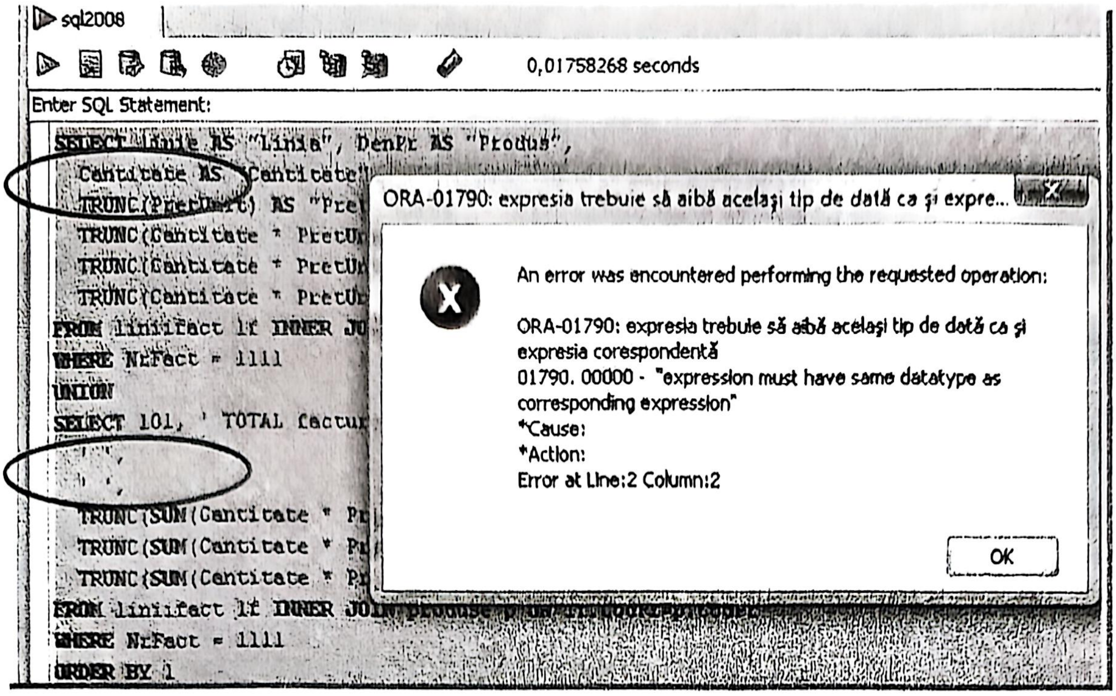


Figura 6.57. Nepotrivire (de tip) la reuniune

Care este valoarea medie a prețului (inclusiv TVA) la care a fost vândută o unitate din „Produs 1"?

Nu, n-am trecut la funcția AVG fără să vă fi anunțat din timp. Soluția se bazează pe raportul dintre suma valorilor şi cantitatea însumată pentru acest produs (figura 6.58). Iată şi interogarea:

```
SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/SUM(Cantitate)
        AS "Pret Unitar Mediu (tare)"
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE DenPr = 'Produs 1'
```

Pret Unitar Mediu (tare)
1131,04462242562929061784897025171624714

Figura 6.58. Preț mediu calculat prin raportul dintre suma valorii şi suma cantităților

Ne bazăm pe faptul că este vorba despre acelaşi produs, deci are sens să cumulăm cantitățile.

Care este situația încasării facturii 1120?

Rezolvarea acestei probleme presupune afişarea, pentru factura 1120, atât a valorii totale, obținută din tabelele LINIIFACT şi PRODUSE, cât şi a valorii încasate, obținută din ÎNCASFACT. Prin urmare, pare firesc să încercăm varianta:

```
SELECT '1120' AS NrFact,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat,
        SUM(Transa) AS Incasat
FROM liniifact lf
```

```
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE F.NrFact = 1120
```

NRFACT	FACTURAT	INCASAT
1120	97664	7315

Figura 6.59. Valorile facturată şi încasată ale facturii 1120

Rezultatul din figura 6.59 arată că lucrurile ar fi în regulă. Dar dacă înlocuim numărul facturii 1120 cu 1111, fraza:

```
SELECT '1111' AS NrFact,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat,
        SUM(Transa) AS Incasat
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE F.NrFact = 1111
```

va genera rezultatul din figura 6.60.

NRFACT	FACTURAT	INCASAT
1111	4346037,5	161988

Figura 6.60. Valoarea facturată şi valoarea încasată (greşită!) pentru factura 1111

Valoarea încasată, 161988, e complet anapoda, valoarea încasată „reală" din această factură fiind 53996. Interesant este că valoarea greşită este de trei ori mai mare decât cea facturată. Care e explicaţia? Răspunsul e mai uşor de aflat dacă vizualizăm rezultatul unei interogări „ajutătoare" ce generează tabela pe care se aplică cele două funcţii SUM (vezi figura 6.61).

```
SELECT lf.*, Cantitate * PretUnit * (1 + ProcTVA) AS Val_fact_linie,
        Transa AS Incasat
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE F.NrFact = 1111
```

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	VAL_FACT_LINIE	INCASAT
1111	1	1	50	1000	59500	53996
1111	2	2	75	1050	85837,5	53996
1111	3	5	500	7060	4200700	53996

Figura 6.61. Rezultatul joncţiunii „componentelor" *facturare* şi *încasare* pentru factura 1111

Întâmplarea face ca pentru această factură să existe o singură tranşă de încasare. Tranşa de 53996 se repetă pentru fiecare linie a facturii 1111. Normal că funcţia

SUM întoarce o valoare triplă faţă de cea reală. Ce-i de făcut? Să încercăm un artificiu. Împărţim sumă tranşelor încasate la numărul liniilor facturii:

```
SELECT '1111' AS NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat,
       SUM(Transa)/COUNT(*) AS Incasat
FROM liniifact lf
       INNER JOIN produse p ON lf.CodPr = p.CodPr
       INNER JOIN facturi f ON lf.NrFact = f.NrFact
       INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE F.NrFact = 1111
```

Merge! Sau, cel puţin, aşa arată figura 6.62.

NRFACT	FACTURAT	INCASAT
1111	4346037,5	53996

Figura 6.62. Valorile (încurajatoare) corecte pentru factura 1111

Rămâne însă un semn de întrebare: ce se întâmplă cu facturile încasate în mai multe tranşe? Spre exemplu, factura 1117 are două linii şi e încasată în trei tranşe. Dacă utilizăm interogarea „ajutătoare" (penultima), schimbând, bineînţeles, numărul facturii, echivalenta tabelei din figura 6.61 se prezintă ca în figura 6.63.

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	VAL_FACT_LINIE	INCASAT
1117	1	2	100	1000	109000	9754
1117	1	2	100	1000	109000	9754
1117	1	2	100	1000	109000	3696
1117	2	1	100	950	113050	9754
1117	2	1	100	950	113050	9754
1117	2	1	100	950	113050	3696

Figura 6.63. Valori repetate datorită mai multor tranşe de încasare – factura 1117

Există nu mai puţin de 3 (tranşe de încasare - ÎNCASFACT) * 2 (linii în LINIIFACT) = 6 linii. Pe baza acestei observaţii ochiometrice, putem să încercăm o soluţie finală, împărţind valoarea facturată la numărul tranşelor de încasare şi valoarea încasată la numărul de linii ale facturii respective:

```
SELECT '1117' AS NrFact,
       SUM(Cantitate * PretUnit * (1 + ProcTVA))/COUNT(DISTINCT i.CodInc) AS
Facturat,
       SUM(Transa)/COUNT(DISTINCT lf.Linie) AS Incasat
FROM liniifact lf
       INNER JOIN produse p ON lf.CodPr = p.CodPr
       INNER JOIN facturi f ON lf.NrFact = f.NrFact
       INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE f.NrFact = 1117
```

NRFACT	FACTURAT	INCASAT
1117	222050	23204

Figura 6.64. Valorile facturată şi încasată (corecte) pentru factura 1117

Rezultatul din figura 6.64 ne îndreptățește să sperăm că soluția e bună. Și, într-adevăr, este funcțională, dar numai pentru facturile care au cel puțin o tranșă de încasare! Pentru cealaltă categorie avem nevoie de cunoștințe ceva mai avansate (joncțiune externă - vezi capitolul 8). În altă ordine de idei, dacă SQL-ul ar fi fost atât de încâlcit la asemenea probleme nu prea complicate, numărul utilizatorilor ar fi fost cu mult mai mic. Noroc de opțiunile pe care le vom prezenta în capitolele următoare ce ne fac viața suportabilă (cel puțin, în SQL).

6.7.3. AVG

Funcția AVG calculează media aritmetică a unei coloane într-o tabelă oarecare, prin divizarea sumei valorilor coloanei respective la numărul de valori nenule ale acesteia. Nefiind prea multe de adăugat, vom discuta doar două exemple destul de simple.

Care este valoarea (fără TVA) medie a liniilor din factura 1111 (figura 6.65)?

SELECT 'Val. medie (fara TVA) a prod. din fact. 1111' AS Explicatii,
TRUNC(AVG(Cantitate * PretUnit),2) AS ValMedie
FROM liniifact
WHERE NrFact = 1111[15]

EXPLICATII	VALMEDIE
Val. medie (fara TVA) a prod. din fact. 1111	1219583,33

Figura 6.65. Valoarea medie a unei linii din factura 1111

Care este media valorilor (cu TVA) la care a fost vândut „Produs 1"?

SELECT 'Val. medie a vanzarilor Produs 1' AS Explicatii,
ROUND(AVG(Cantitate*PretUnit*(1 + ProcTVA)),2) AS "Valoare (cu TVA) medie"
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'

Adesea, prin aplicarea funcției AVG sau prin formularea unor expresii ce conțin rapoarte între atribute/constante, se obțin rezultate cu numeroase zecimale. În cazul de față, pentru preîntâmpinarea acestui disconfort vizual și intelectual, a fost preferată funcția ROUND ce rotunjește media la două zecimale (vezi figura 6.66)[16].

Explicatii	Valoare (cu TVA) medie
Val. medie a vinzarilor Produs 1	109837.000000

Figura 6.66. Media valorilor (cu TVA) la care a fost vândut, într-o factură, Produs 1

15. Sintaxă Oracle/PostgreSQL. În DB2 și SQL Server, funcția TRUNC se folosește așa cum a fost arătat la interogarea ce obține liniile din figura 6.53.
16. Pare dubios că, deși facem rotunjirea la două zecimale, coloana medie are șase zecimale. Încă o dată, este o chestiune doar de afișare, calculul făcându-se, într-adevăr, la precizie de două zecimale. Dacă nu mă credeți, încercați cu produsele 2 sau 5.

6.7.4. MIN & MAX

Funcţiile MIN şi MAX sunt deosebit de utile în diverse tipuri de analiză a datelor, determinând valoarea maximă, respectiv minimă, pentru o coloană (atribut). Se pot folosi şi pentru atribute de tip şir de caractere, caz în care elementul de comparaţie este codul ASCII/UTF al caracterelor.

Care este localitatea cu ultima denumire, în ordine alfabetică?

Atributul care ne interesează este Loc din tabela CODURI_POŞTALE, iar valoarea determinată este Vaslui (figura 6.67).

```
SELECT MAX(Loc) AS "Ultima_Localitate_Alfabetic"
FROM coduri_postale
```

Ultima_Localitate_Alfabetic
Vaslui

Figura 6.67. (A fost sau n-a fost) Vaslui, ultima localitate (doar din punct de vedere alfabetic!)

Care sunt primul client şi ultimul client (în ordinea numelui) din judeţul Iaşi (figura 6.68)?

Nu e nicio problemă dacă o clauză SELECT conţine două sau mai multe funcţii-agregat, atâta vreme cât nu apar alte coloane (neagregat).

```
SELECT MIN(DenCl) AS Primul_Client,
       MAX(DenCl) AS Ultimul_Client
FROM clienti c
       INNER JOIN coduri_postale cp ON c.codpost = cp.codpost
       INNER JOIN judete j ON cp.jud = j.jud
WHERE Judet = 'Iasi'
```

PRIMUL_CLIENT	ULTIMUL_CLIENT
Client 1 SRL	Client 4

Figura 6.68. Primul şi ultimul client (d.p.d.v. alfabetic) din judeţul Iaşi

Care este cel mai mare preţ unitar (fără TVA) la care a fost vândut Produs 1?

```
SELECT MAX(PretUnit)
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
```

MAX(PRETUNIT)
1000

Figura 6.69. Preţul unitar maxim la care a fost vândut Produsul 1

Din păcate, dacă dorim să aflăm şi în ce factură produsul are preţul unitar maxim, soluţia:

SELECT MAX(PretUnit), NrFact
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'

nu funcționează (vezi figura 6.70), deoarece în clauza SELECT apare o funcție-agregat împreună cu un atribut „simplu" în lipsa clauzei de grupare (vezi capitolul următor).

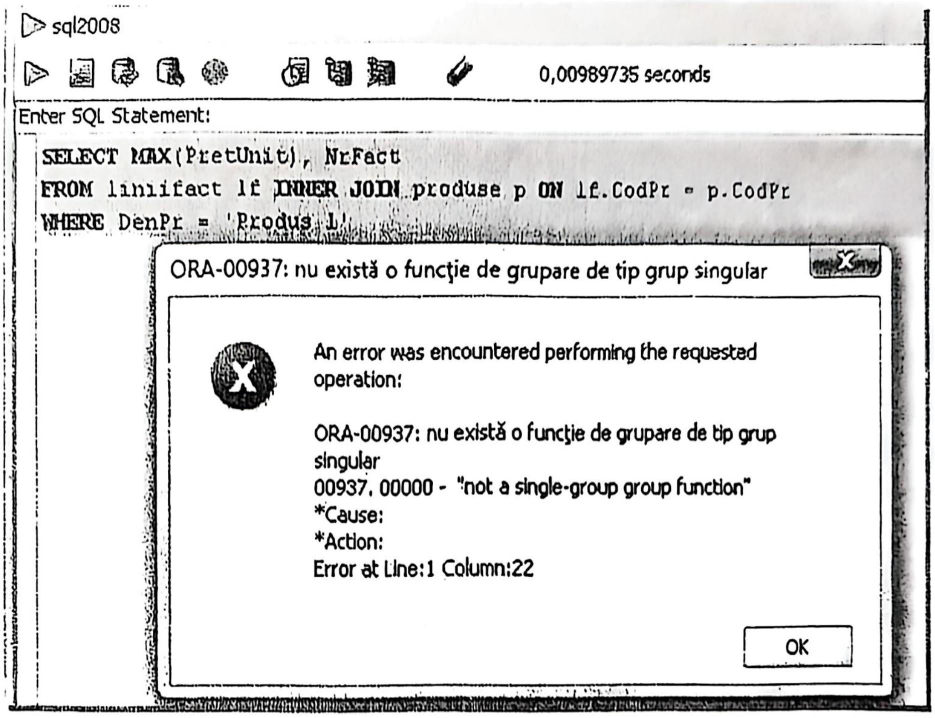


Figura 6.70. În lipsa clauzei GROUP BY este interzisă apariția în clauza SELECT a unei funcții-agregat împreună cu un atribut/expresie neagregat

Până la subconsultările din capitolul 9 încercăm o soluție-surogat Oracle/ PostgreSQL, concatenând prețul cu numărul facturii, plus ceva text, iar funcția MAX o vom aplica la acest șir:

SELECT MAX('Pretul maxim = ' || LF.PretUnit ||
', apare in factura ' || NrFact)
AS Mesaj_cam_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'[17]

MESAJ_CAM_LUNG
Pretul maxim=950, apare in factura 3117

Figura 6.71. Cârpeli Oracle (vol.1)

Rezultatul din figura 6.71 este eronat, deoarece produsul 1 s-a vândut la un preț mai mare decât 950 (1000 de lei, în facturile 1111, 2111 și 3111), după cum se vede în figura 6.72, al cărei conținut a fost extras prin interogarea:

17. În MS SQL Server, interogarea are forma: SELECT MAX('Pretul maxim = ' + CAST (LF.PretUnit AS VARCHAR) + ', apare in factura ' + CAST(NrFact AS VARCHAR)) AS Mesaj_cam_lung FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr WHERE DenPr = 'Produs 1'.

```
SELECT lf.*
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
ORDER BY pretunit DESC
```

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1111	1	1	50	1000
2111	1	1	57	1000
3111	1	1	57	1000
1117	2	1	100	950
3117	2	1	110	950
2117	2	1	110	950
1118	2	1	150	930
2118	2	1	120	930
3118	2	1	120	930

Figura 6.72. Cârpeli (volumul 1)

Ne întrebăm, fireşte, de la ce ni se trage ruşinea din figura 6.71. Ei bine, de la concatenare! Pentru a înţelege mai bine, să introducem şirul de caractere ce constituie argumentul funcţiei MAX chiar în interogarea de mai sus:

```
SELECT lf.*, 'Pretul maxim = ' || LF.PretUnit ||
        ', apare in factura '|| NrFact AS Mesaj_la_fel_de_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
ORDER BY pretunit DESC
```

De data aceasta, figura 6.73 este destul de limpede pentru a înţelege modalitatea de ordonare. Practic, la concatenare se ia în calcul prima cifră semnificativă a preţului unitar, astfel încât 9 (de la 950) este mai mare decât 1 (de la 1000).

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	MESAJ_LA_FEL_DE_LUNG
3117	2	1	110	950	Pretul maxim=950, apare in factura 3117
2117	2	1	110	950	Pretul maxim=950, apare in factura 2117
1117	2	1	100	950	Pretul maxim=950, apare in factura 1117
3118	2	1	120	930	Pretul maxim=930, apare in factura 3118
2118	2	1	120	930	Pretul maxim=930, apare in factura 2118
1118	2	1	150	930	Pretul maxim=930, apare in factura 1118
3111	1	1	57	1000	Pretul maxim=1000, apare in factura 3111
2111	1	1	57	1000	Pretul maxim=1000, apare in factura 2111
1111	1	1	50	1000	Pretul maxim=1000, apare in factura 1111

Figura 6.73. La concatenare, se ia în calcul prima cifră semnificativă a preţului

Pentru a rezolva problema, forţăm conversia preţului unitar în şir de caractere pe un număr prestabilit de poziţii. Funcţia PostgreSQL/Oracle TO_CHAR este cea care va asigurarea „umplerea" zerourilor nesemnificative cu spaţii şi, astfel, viabilitatea ideii:

```
SELECT lf.*, 'Pretul maxim=' || TO_CHAR(LF.PretUnit, '999999.99')
        || ', apare in factura '|| NrFact AS Mesaj_la_fel_de_lung
```

FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
ORDER BY Mesaj_la_fel_de_lung DESC

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	MESAJ_LA_FEL_DE_LUNG
3111	1	1	57	1000	Pretul maxim= 1000.00, apare in factura 3111
2111	1	1	57	1000	Pretul maxim= 1000.00, apare in factura 2111
1111	1	1	50	1000	Pretul maxim= 1000.00, apare in factura 1111
3117	2	1	110	950	Pretul maxim= 950.00, apare in factura 3117
2117	2	1	110	950	Pretul maxim= 950.00, apare in factura 2117
1117	2	1	100	950	Pretul maxim= 950.00, apare in factura 1117
3118	2	1	120	930	Pretul maxim= 930.00, apare in factura 3118
2118	2	1	120	930	Pretul maxim= 930.00, apare in factura 2118
1118	2	1	150	930	Pretul maxim= 930.00, apare in factura 1118

Figura 6.74. Folosirea funcției TO_CHAR pentru ordonarea corectă a prețului

Ordinea este acum cea din figura 6.74. În MS SQL Server, în locul funcției TO_CHAR se foloseşte STR:

```
SELECT MAX('Pretul maxim=' + STR(LF.PretUnit,10,2) +
        ', apare in factura ' + STR(NrFact,10)) AS Mesaj_cam_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
```

Fireşte, în loc de TO_CHAR, puteam recurge la eleganta CAST. Lucrurile nu sunt chiar aşa de simple, întrucât ordinea rândurilor din rezultatul interogării[18]:

```
SELECT lf.*, 'Pretul maxim=' || CAST (LF.PretUnit AS CHAR(12))
        || ', apare in factura '|| NrFact AS Mesaj_la_fel_de_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
ORDER BY Mesaj_la_fel_de_lung DESC
```

arată la fel de deprimant ca în figura 6.73. Remedierea situației presupune o amețitoare combinație LPAD(TRIM(...)) prin care forțăm alinierea la dreapta a prețului unitar şi completarea cu spații la stânga pentru umplerea eventualelor poziții datorate zerourilor nesemnificative:

```
SELECT lf.*, 'Pretul maxim = ' || LPAD(
        TRIM (TRAILING FROM
                        CAST(LF.PretUnit AS CHAR(12))
                                )
                        12, ' ')
        || ', apare in factura '|| NrFact AS Mesaj_la_fel_de_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
ORDER BY Mesaj_la_fel_de_lung DESC
```

În fine, ajungem şi la final, considerând expresia concatenată argument al funcției MAX:

18. Sintaxă Oracle/PostgreSQL. Valabil şi pentru următoarele două interogări.

```
SELECT MAX ('Pretul maxim = ' ||
        LPAD(
        TRIM (TRAILING FROM CAST(LF.PretUnit AS CHAR(12))
                    ), 12, ' ')
        || ', apare in factura '|| NrFact) AS PretUnitar_Maxim
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
```

Soluţia funcţionează, cel puţin dacă ne luăm după rezultatul din figura 6.75, aşa că n-are de ce să ne fie jenă (prea tare) de improvizaţie. Rezultatul este incomplet, însă, atunci când preţul maxim apare în două sau mai multe facturi, deoarece interogarea de mai sus extrage numai una dintre facturi.

PRETUNITAR_MAXIM
Pretul maxim=   1000, apare in factura 3111

Figura 6.75. Preţul maxim pentru Produs 1 şi factura în care apare acest preţ

Trebuie spus, însă, că, în DB2, versiunea iniţială a interogării - cea care foloseşte funcţia CAST funcţionează ireproşabil datorită faptului că, la conversie, se completează implicit cu zerouri nesemnificative (vezi figura 6.76):

```
SELECT MAX('Pretul maxim = ' || CAST( LF.PretUnit AS CHAR(10)) ||
        ', apare in factura '|| CAST ( NrFact AS CHAR(10)))
        AS Mesaj_cam_lung
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 1'
```

MESAJ_CAM_LUNG
Pretul maxim=0001000.00, apare in factura 00003111.

Figura 6.76. Varianta DB2 de afişare a preţului maxim pentru Produs 1 şi a facturii în care apare acest preţ

Care este cel mai mare şi cel mai mic preţ unitar (fără TVA) la care a fost vândut Produs 2?

Suntem plini ochi de know-how-ul de la problema anterioară, aşa că ne distrăm un pic (rezultatul fiind cel din figura 6.77):

```
SELECT MAX( 'Pret maxim=' ||
        LPAD(
        TRIM (TRAILING FROM CAST(LF.PretUnit AS CHAR(12))), 12, ' ')
          || ', factura:' || CAST (NrFact AS CHAR(10)))
                        AS Max_Pret_Produs_2,
        MIN ('Pret minim = ' ||
        LPAD(
        TRIM (TRAILING FROM CAST(LF.PretUnit AS CHAR(12))), 12, ' ')
        || ', factura:' || CAST (NrFact AS CHAR(10)))
        AS Min_Pret_Produs_2
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DenPr = 'Produs 2'
```

MAX_PRET_PRODUS_2	MIN_PRET_PRODUS_2
Pret maxim= 1120, factura:1120	Pret minim= 925, factura:1115

Figura 6.77. Prețurile maxime și minime ale Produs 2 și facturile în care apar aceste prețuri

Din nou trebuie să precizăm că, în cazul în care cele două prețuri s-au înregistrat în mai multe facturi, interogarea nu indică decât una dintre acestea și, în plus, interogarea funcționează ca atare în PostgreSQL și Oracle.

Care sunt cele mai mari două prețuri unitare (fără TVA) la care a fost vândut Produs 2?

Cu remarca răutăcioasă legată de incompletitudinea rezultatului, profităm de problemă pentru a autojoncționa tabela LINIIFACT. Atenție la predicatul din INNER JOIN (avem un alt exemplu de theta-joncțiune)!

```
SELECT 'Produs 2: '||
        MAX('Primul PU: '||TO_CHAR(LF1.PretUnit,'999999.99')||
        ', al doilea PU: '||TO_CHAR(LF2.PretUnit,'99999999')
        ||' - factura primului:'||LF1.NrFact||
        ', factura-al doilea:'||LF2.Nrfact)
                AS "Cele mai mari doua PretUnit"
FROM liniifact LF1
        INNER JOIN liniifact LF2
                ON LF1.CodPr = LF2.CodPr AND LF1.PretUnit > LF2.PretUnit
        INNER JOIN produse P ON LF1.CodPr = P.CodPr AND DenPr = 'Produs 2'
```

Cele mai mari doua PretUnit
Produs 2: Primul PU: 1120.00, al doilea PU: 1100 - factura primului:1120, factura-al doilea:3118

Figura 6.78. Cele mai mari două prețuri la care s-a vândut Produs 2
și facturile în care apar aceste prețuri

Acestei variante îi putem reproșa (ca multor altora din această carte) că este prea oraclisto-postgresqlistă, așa că putem să formulăm o variantă mai generală, executabilă și în DB2:

```
SELECT 'Produs 2: '|| MAX('Primul PU: '|| CAST (LF1.PretUnit AS CHAR(10)) ||
        ', al doilea PU: '|| CAST (LF2.PretUnit AS CHAR(10))
        ||' - factura primului:'||CAST (LF1.NrFact AS CHAR(8))||
        ', factura-al doilea:'|| CAST (LF2.Nrfact AS CHAR(8)))
                AS "Cele mai mari doua PretUnit"
FROM liniifact LF1
        INNER JOIN liniifact LF2
                ON LF1.CodPr = LF2.CodPr AND LF1.PretUnit > LF2.PretUnit
        INNER JOIN produse P ON LF1.CodPr = P.CodPr AND DenPr = 'Produs 2'
```

Care este situaţia încasării facturii 1117?

Reluăm ultima variantă de rezolvare a acestei probleme, introducând acum şi funcţia MAX:

```
SELECT '1117' AS NrFact,
        SUM(Cantitate * PretUnit * (1+ProcTVA)) / COUNT(DISTINCT i.CodInc)
        AS Facturat,
        SUM(Transa) / MAX(lf.Linie) AS Incasat
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
WHERE f.NrFact = 1117
```

Capitolul 7
Grupări

Limbajul SQL, ca şi algebra relaţională din care se trage, este orientat preponderent pe tupluri, lucrând cu seturi de linii ale tabelelor, şi nu cu fiecare linie în parte. Funcţiile-agregat pe care le-am discutat în paragraful anterior operau asupra tuturor liniilor (tuplurilor) tabelei din clauza FROM care îndeplineau condiţia de filtrare din clauza WHERE (când în clauza FROM apar două sau mai multe tabele joncţionate, liniile sunt filtrate atât prin clauza WHERE, cât şi prin clauza ON de la INNER JOIN).

Folosing clauza GROUP BY, funcţiile-agregat pot fi aplicate la nivel de grup de înregistrări, rezultatul având atâtea linii câte valori distincte are atributul/coloana de grupare. Mai mult, putem elimina o parte din grupuri folosind un predicat specificat într-o clauză specială - HAVING.

Dacă în capitolele precedente am fost destul de democraţi în materie de sintaxă cu toate cele patru servere BD, în acesta vom privilegia variantele cele mai conforme cu standardul, adică cele Oracle şi PostgreSQL. Pe baza discuţiilor de până acum, veţi putea porta interogările pe sintaxa DB2 şi SQL Server fără prea multe dureri de cap.

7.1. Gruparea după un criteriu

Clauza GROUP BY formează grupe (grupuri) de tupluri ale unei relaţii, pe baza valorilor comune ale unui atribut. Rezultatul unei fraze SELECT ce conţine această clauză se obţine prin regruparea tuturor liniilor din tabelele enumerate în FROM, extrăgându-se câte o apariţie pentru fiecare valoare distinctă a coloanei/coloanelor de grupare. Cel mai simplu format este:

```
SELECT expresie 1, expresie 2, ..., expresie m
FROM tabelă
WHERE predicat
GROUP BY coloană-de-grupare
```

Să luăm câteva exemple pentru a clarifica discuţia.

Care este valoarea fără TVA a facturilor cu numere cuprinse între 1111 şi 1119?

Valoarea (fără TVA a) unei facturi se obţine însumând rezultatele produsului *Cantitate * PreţUnit* pentru toate liniile din factura respectivă. Totalizarea se face pe facturi, astfel atributul de grupare fiind NrFact:

```
SELECT NrFact, SUM(Cantitate*PretUnit) as ValFaraTVA
FROM liniifact
```

WHERE NrFact BETWEEN 1111 AND 1119

GROUP BY NrFact

Lansarea în execuţie a acestei interogări declanşează următoarea succesiune de operaţii:

a) din tabela LINIIFACT se extrag numai tuplurile în care valorile atributului NrFact sunt cuprinse între 1111 şi 1119;
b) se constituie câte un grup pentru fiecare valoare distinctă a NrFact; în cazul nostru, deoarece sunt nouă valori distincte ale NrFact selectate, vor fi constituite nouă grupuri;
c) în interiorul fiecărui grup (care are un număr de rânduri egal cu numărul liniilor din factura corespunzătoare grupului) se execută funcţia-agregat – *SUM(Cantitate * PreţUnit)*;
d) se obţine rezultatul al cărui număr de linii coincide cu valorile distincte ale NrFact. Ordinea liniilor nu este garantată în lipsa clauzei ORDER BY.

Paşii b), c), şi d) sunt schematizaţi (sau sugeraţi) în figura 7.1. Deşi valorile atributului de grupare sunt ordonate crescător în figură, acest lucru nu este garantat automat la orice grupare.

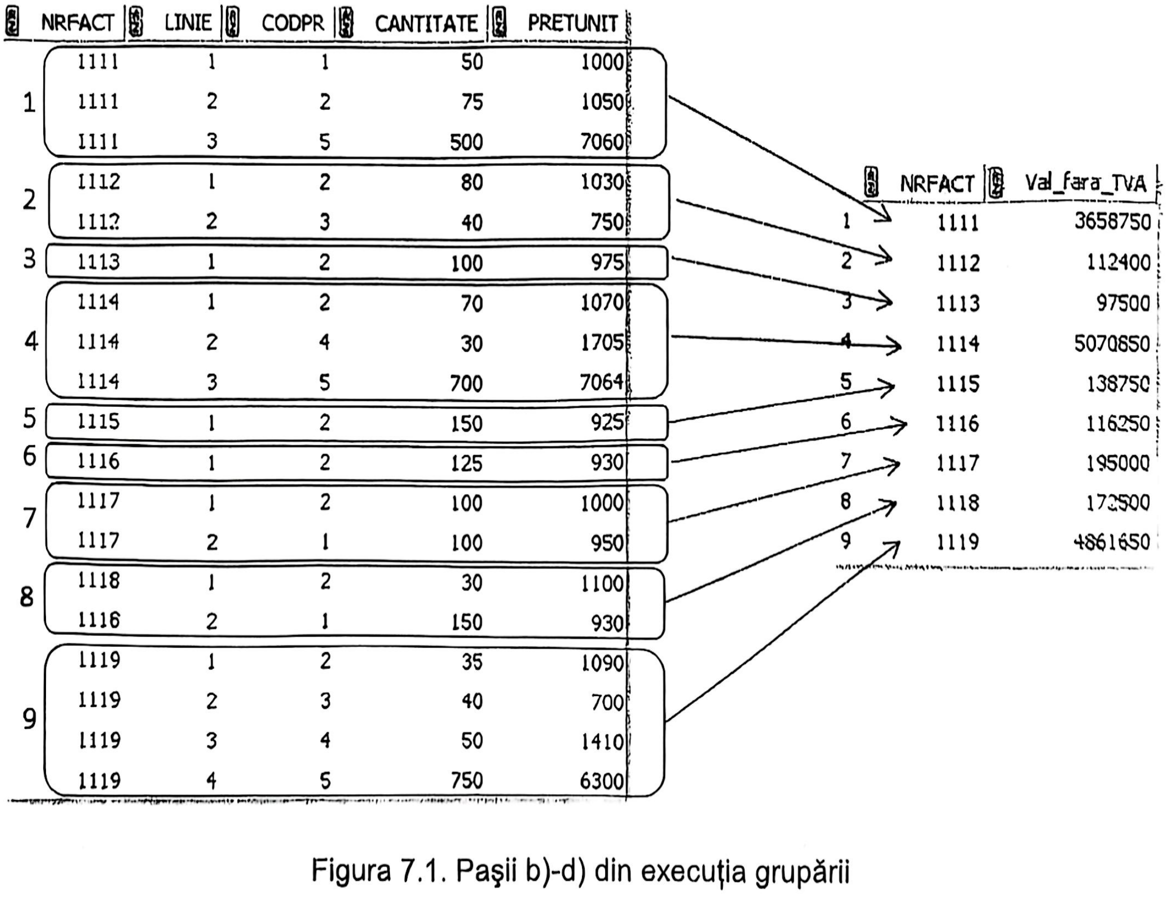


Figura 7.1. Paşii b)-d) din execuţia grupării

Care este numărul de facturi emise în fiecare zi cu vânzări?

O zi cu vânzări este o dată în care avem măcar o factură întocmită. Întrucât în enunţul problemei apare ...*în fiecare zi cu vânzări,* atributul care corespunde zilelor de vânzare este DataFact din FACTURI. Acesta va fi atributul de grupare. Numărul de facturi se obţine cu ajutorul funcţiei COUNT:

```
SELECT DataFact AS "Zi", COUNT(*) AS "Numar facturi"
FROM facturi
GROUP BY DataFact
```

Zi	Numar facturi
01-08-2007	4
04-08-2007	1
17-09-2007	1
03-08-2007	1
01-09-2007	2
07-10-2007	1
02-08-2007	2
21-08-2007	2
07-08-2007	4
14-08-2007	3
15-08-2007	2
16-08-2007	2
02-09-2007	2
22-08-2007	1
10-09-2007	2

Figura 7.2. Numărul de facturi întocmite în fiecare zi cu vânzări

După cum se observă în figura 7.2, în Oracle, ordinea extragerii liniilor în rezultat nu este cea a valorilor atributului de grupare. La fel stau lucrurile în PostgreSQL. De aceea, pentru a fi siguri de ordonare, trebuie să folosim clauza ORDER BY. Ne interesează ca zilele să apară în ordine crescătoare:

Zi	Numar facturi
01-08-2007	4
02-08-2007	2
03-08-2007	1
04-08-2007	1
07-08-2007	4
14-08-2007	3
15-08-2007	2
16-08-2007	2
21-08-2007	2
22-08-2007	1
01-09-2007	2
02-09-2007	2
10-09-2007	2
17-09-2007	1
07-10-2007	1

Figura 7.3. Ordonarea valorilor atributului de grupare

```
SELECT DataFact AS "Zi", COUNT(*) AS "Numar facturi"
FROM facturi GROUP BY DataFact ORDER BY DataFact
```

Rezultatul este cel din figura 7.3. DB2 şi SQL Server asigură ordonarea liniilor în rezultat, chiar şi fără clauza ORDER BY.

Care este valoarea totală a vânzărilor pentru fiecare zi din luna septembrie 2007 în care s-au emis facturi (figura 7.4)?

Dacă atributul de grupare este tot DataFact, pentru că ne interesează valoarea *zilnică*, obţinerea vânzărilor (cu tot cu TVA) reclamă joncţionarea tabelei FACTURI cu LINIIFACT şi apoi cu PRODUSE:

```
SELECT DataFact, TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValTotala
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DataFact
ORDER BY 1
```
[1]

DATAFACT	VALTOTALA
01-09-2007	4596401
02-09-2007	238437
10-09-2007	424704
17-09-2007	179565

Figura 7.4. Vânzările zilnice pentru luna septembrie 2007

Care sunt, în luna august 2007, numărul de facturi şi valoarea vânzărilor pentru fiecare client?

Gruparea trebuie făcută după client. În acest sens putem folosi orice cheie primară sau alternativă a tabelei CLIENŢI. Fireşte, mai nimerită este denumirea clientului, aşa încât clauza de grupare va fi *GROUP BY DenCl*. Extragerea vânzărilor corespunzătoare fiecărui client presupune „filiera" de joncţionare CLIENŢI-FACTURI-LINIIFACT-PRODUSE.

```
SELECT DenCl, COUNT(DISTINCT F.NrFact) AS NrFacturilor,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValTotala
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
```

1. Sintaxă PostgreSQL/Oracle.

AND EXTRACT (MONTH FROM DataFact) = 8

GROUP BY DenCl

ORDER BY 1[2]

Ei bine, rezultatul are o ciudățenie. În partea stângă a figurii 7.5 avem interogarea de mai sus și rezultatul său. În partea dreaptă avem o altă interogare ce calculează, la rândul său, numărul facturilor emise fiecărui client în august 2007, dar fără a mai apela la LINIIFACT și PRODUSE, întrucât ne interesează numai numărul facturilor, nu și valoarea acestora.

```
SELECT DenCl,
  COUNT(DISTINCT F.NrFact) AS NrFacturilor_v1,
  TRUNC(
         SUM(Cantitate * PretUnit * (1+ProcTVA))
         ,0 ) AS ValTotala
FROM clienti c
 INNER JOIN facturi f ON c.CodCl=f.CodCl
 INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
 INNER JOIN produse p ON lf.CodPr=p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
  AND EXTRACT (MONTH FROM DataFact) = 8
GROUP BY DenCl
ORDER BY 1
```

DENCL	NRFACTURILOR	VALTOTALA
Client 1 SRL	9	10040251
Client 2 SA	2	233805
Client 3 SRL	2	11594166
Client 4	2	9479109
Client 5 SRL	2	278958
Client 6 SA	1	6021706
Client 7 SRL	2	263562

```
SELECT DenCl,
  COUNT(DISTINCT F.NrFact) AS NrFacturilor_v2
FROM clienti c
 INNER JOIN facturi f ON c.CodCl=f.CodCl
WHERE EXTRACT (YEAR FROM DataFact) = 2007
  AND EXTRACT (MONTH FROM DataFact) = 8
GROUP BY DenCl
ORDER BY 1
```

DENCL	NRFACTURILOR_V2
Client 1 SRL	9
Client 2 SA	2
Client 3 SRL	2
Client 4	2
Client 5 SRL	4
Client 6 SA	1
Client 7 SRL	2

Figura 7.5. Facturi pierdute „pe drum"

Nedumerirea apare la numărul facturilor corespunzătoare Clientului 5 SRL. Dacă în prima variantă sunt 2, în a doua apar cu 2 în plus. Încercăm să localizăm problema și să extragem numai facturile acestui client, prima dată doar prin joncțiunea CLIENȚI-FACTURI, a doua oară prin joncțiunea CLIENȚI-FACTURI--LINIIFACT (vezi figura 7.6).

Comparând cele două jumătați ale figurii găsim dezlegarea enigmei: două dintre facturi, 1122 și 2122, nu conțin nicio linie. Astfel, la joncționarea cu LINIIFACT, aceste facturi dispar și, de aici, diferența de 2 din figura 7.5. Așadar, în figura 7.5, ambele numere furnizate sunt corecte, numai că reprezintă lucruri diferite: primul (cel din stânga) furnizează numărul facturilor „corecte", al doilea numărul total de facturi, indiferent dacă acestea au sau nu vreo linie.

2. Sintaxă PostgreSQL/Oracle.

```
SELECT f.*
FROM clienti c
    INNER JOIN facturi f ON c.CodCl=f.CodCl
WHERE EXTRACT (YEAR FROM DataFact) = 2007
  AND EXTRACT (MONTH FROM DataFact) = 8
  AND DenCl='Client 5 SRL'
ORDER BY NrFact
```

NRFACT	DATAFACT	CODCL	OBS
1112	01-08-2007	1005	Probleme cu transportul
1122	07-08-2007	1005	(null)
2112	14-08-2007	1005	Probleme cu transportul
2122	22-08-2007	1005	(null)

```
SELECT lf.*
FROM clienti c
  INNER JOIN facturi f ON c.CodCl=f.CodCl
  INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007
  AND EXTRACT (MONTH FROM DataFact) = 8
  AND DenCl='Client 5 SRL'
ORDER BY Nrfact
```

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
1112	1	2	80	1030
1112	2	3	40	750
2112	1	2	85	1030
2112	2	3	65	750

Figura 7.6. Două dintre facturi nu au nicio linie

Care este restul de încasat pentru fiecare factură?

Aşa cum reiese din figura 7.7, soluţia următoare oferă un rezultat incomplet. Lipsesc facturile care nu au nicio tranşă de încasare. Pentru rezolvarea cazului avem nevoie de clauza HAVING, dar şi de joncţiunea externă, după cum vom vedea în capitolul următor.

```
SELECT f.NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA))/
        COUNT(DISTINCT i.CodInc) AS Facturat,
        SUM(Transa)/MAX(lf.Linie) AS Incasat
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
GROUP BY f.NrFact[3]
```

NRFACT	FACTURAT	INCASAT
1111	4346037,5	53996
1112	125516	125516
1113	106275	106275
1117	222050	23204
1118	201975	201975
1120	97664	7315

Figura 7.7. Valorile facturată şi încasată pentru fiecare factură – rezultat incomplet

Care sunt vânzările, cantitativ şi valoric, pentru fiecare produs?

De data aceasta, gruparea se face după valorile atributului DenPr, iar în joncţiune nu avem nevoie decât de LINIIFACT şi PRODUSE:

3. Sintaxă valabilă în toate cele patru servere BD.

```
SELECT DenPr,
        SUM(Cantitate) AS Cantitativ,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Valoric
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr
```
[4]

Rezultatul este cel din figura 7.8.

DENPR	CANTITATIV	VALORIC
Produs 3	290	251685
Produs 2	2554	2784023
Produs 4	190	301657
Produs 1	874	988533
Produs 5	5580	44844436

Figura 7.8. Vânzările cantitative şi valorice pe produse

7.2. Gruparea după două sau mai multe criterii

O informaţie esenţială care lipseşte din figura 7.8 este unitatea de măsură, fără de care nu putem fi siguri dacă totatul cantitativ se referă la cutii, sticle, pachete, baxuri etc. Din păcate, varianta:

```
SELECT DenPr, UM,
        SUM(Cantitate) AS Cantitativ,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Valoric
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr
```

nu funcţionează (vezi figura 7.9), deoarece unitatea de măsură (UM) apare separat de atributul de grupare, nefiind inclus în funcţia/funcţiile care se execută la nivelul grupului.

Există însă două remedii cât se poate de simple. Mai întâi, se poate concatena unitatea de măsură cu denumirea produsului şi se poate efectua gruparea după această coloană-şir:

4. Sintaxă DB2, Oracle şi PostgreSQL.

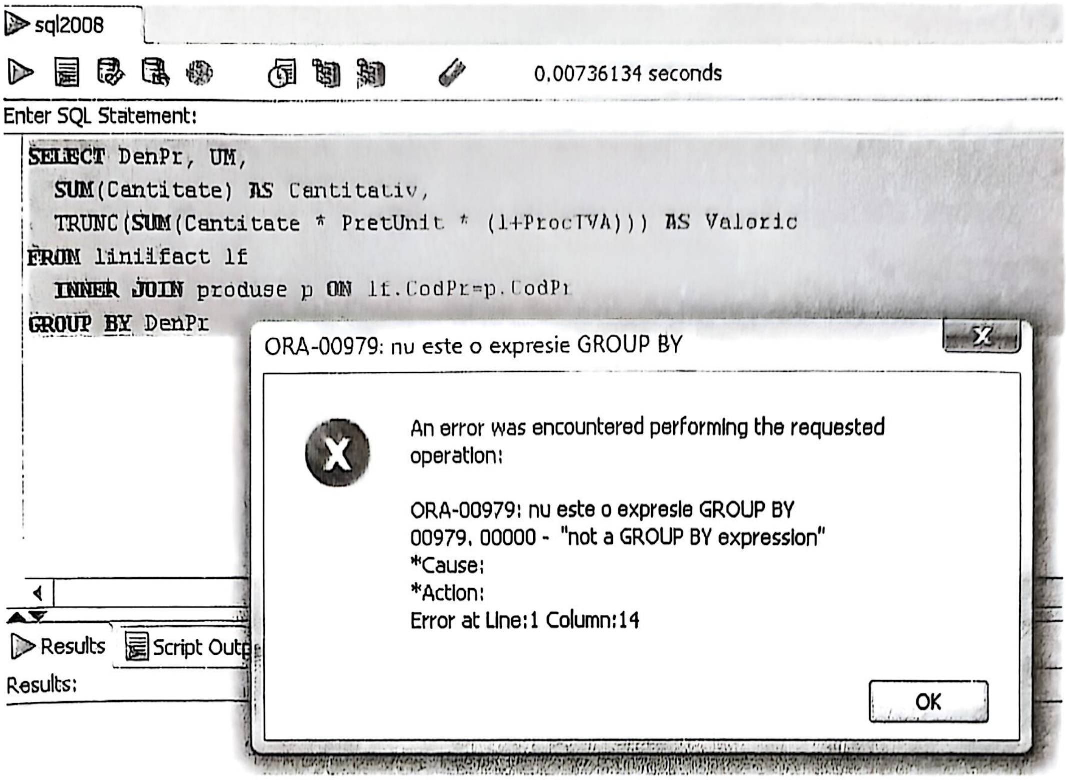


Figura 7.9. Eroare de grupare

```
SELECT DenPr || ' (' || UM || ')' AS "Produs (UM)",
        SUM(Cantitate) AS Cantitativ,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Valoric
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr || ' (' || UM || ')'
ORDER BY 1
```
[5]

cu rezultatul în figura 7.10.

Produs (UM)	CANTITATIV	VALORIC
Produs 1 (buc)	874	988533
Produs 2 (kg)	2554	2784023
Produs 3 (kg)	290	251685
Produs 4 (l)	190	301657
Produs 5 (buc)	5580	44844436

Figura 7.10. Gruparea după şirul obţinut prin concatenarea denumirii cu unitatea de măsură

Atenţie! Expresia din GROUP BY -- trebuie să fie identică cu cea din clauza SELECT, altminteri va fi declanşată o eroare (sunt tolerate câteva „deviaţii", dar, pentru moment, luăm de bună această poruncă).

5. Sintaxă DB2, Oracle şi PostgreSQL.

Al doilea remediu ţine de gruparea după două atribute, DenPr şi UM, cele două atribute fiind incluse şi în clauza SELECT (vezi şi figura 7.11):

```
SELECT DenPr, UM,
        SUM(Cantitate) AS Cantitativ,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Valoric
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr, UM
ORDER BY 1
```
[6]

DENPR	UM	CANTITATIV	VALORIC
Produs 1	buc	874	988533
Produs 2	kg	2554	2784023
Produs 3	kg	290	251685
Produs 4	l	190	301657
Produs 5	buc	5580	44844436

Figura 7.11. Gruparea după două atribute

Numărul de grupuri din figura 7.11 este acelaşi cu cel din figura 7.8, întrucât DenPr este cheie alternativă în tabela PRODUSE. Tot datorită faptului că DenPr este cheie alternativă, putem ca, la grupare, să schimbăm ordinea celor două atribute (ne ajută şi faptul că ordonarea se face după denumirea produsului):

```
SELECT DenPr, UM,
        SUM(Cantitate) AS Cantitativ,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Valoric
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY UM, DenPr
ORDER BY 1
```
[7]

Care este situaţia vânzărilor pe clienţi şi zile în luna septembrie 2007?

Ne interesează valoarea facturilor emise pe clienţi şi, pentru fiecare client, cuantumul zilnic al acestora. Este momentul să folosim o veritabilă grupare după două atribute (interogarea funcţionează în Oracle):

```
SELECT DenCl AS DenumireClient, DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
```

6. Sintaxă DB2, Oracle şi PostgreSQL.
7. Sintaxă DB2, Oracle şi PostgreSQL.

```
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
ORDER BY 1,2
```
[8]

Nu cred că mai are rost să insistăm asupra faptului că ordinea atributelor este decisivă pentru modul de grupare şi, implicit, pentru valorile calculate.

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	02-09-2007	110907
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 2 SA	02-09-2007	127530
Client 5 SRL	01-09-2007	153442
Client 7 SRL	10-09-2007	136849

Figura 7.12. Vânzările zilnice pentru fiecare client în luna septembrie 2007

Care este situaţia vânzărilor fiecărui produs pe fiecare regiune?

Cele două atribute de grupare sunt DenPr din tabela PRODUSE şi Regiune din tabela JUDEŢE. Clauza FROM este ceva mai impresionantă, întrucât trebuie joncţionate nu mai puţin de şase tabele: PRODUSE, LINIIFACT, FACTURI, CLIENŢI, CODURI_POŞTALE şi JUDEŢE:

```
SELECT DenPr, Regiune,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr, Regiune
ORDER BY 1,2
```
[9]

8. Sintaxă Oracle şi PostgreSQL.

9. Sintaxă Oracle şi PostgreSQL.

DENPR	REGIUNE	VINZARI
Produs 1	Moldova	988533
Produs 2	Banat	681086
Produs 2	Moldova	2102937
Produs 3	Banat	151725
Produs 3	Moldova	99960
Produs 4	Moldova	301657
Produs 5	Moldova	44844436

Figura 7.13. Vânzările pe regiuni ale fiecărui produs

Să se obțină situația vânzărilor pe produse, regiuni și zile în luna septembrie 2007

Exemplul de față este o bună ocazie de a folosi gruparea după trei atribute, și, implicit, o analiză a vânzărilor pe cele axe tradiționale: sortimentală, teritorială și calendaristică - vezi figra 7.14.

```
SELECT DenPr, Regiune, DataFact AS Zi,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
            AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenPr, Regiune, DataFact
ORDER BY 1,2,3
```

Fără îndoială, interogările de mai sus constituie un ajutor esențial în analiza vânzărilor oricărei firme (chiar și de stat). Ceea ce lipsește este un ingredient important al oricărui raport de acest gen - subtotalul. Vom recurge, așa cum v-ați obișnuit, la improvizații. Reluăm una din problemele anterioare, modificându-i ușor enunțul:

Să se obțină situația vânzărilor pe clienți și zile în luna septembrie 2007, afișându-se câte un subtotal la nivel de client și un total general.

Ideea improvizației este să "alipim", pentru fiecare client, după vânzările zilnice, câte o linie care să conțină subtotalul. Pentru cele două tipuri de rânduri ale raportului scriem două interogări pe care le conectăm prin UNION:

DENPR	REGIUNE	ZI	VINZARI
Produs 1	Moldova	01-09-2007	67830
Produs 1	Moldova	10-09-2007	124355
Produs 1	Moldova	17-09-2007	132804
Produs 2	Banat	01-09-2007	95429
Produs 2	Banat	10-09-2007	136849
Produs 2	Moldova	01-09-2007	90415
Produs 2	Moldova	02-09-2007	238437
Produs 2	Moldova	10-09-2007	163500
Produs 2	Moldova	17-09-2007	46761
Produs 3	Banat	01-09-2007	58012
Produs 5	Moldova	01-09-2007	4284714

Figura 7.14. Gruparea vânzărilor pe produse, regiuni şi zile

```
SELECT DenCl AS DenumireClient,
        DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
UNION
SELECT DenCl || ' - SUBTOTAL',
        NULL,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
ORDER BY 1,2
```

Pentru uni-compatibilitatea rezultatelor celor două SELECT-uri am folosit, în cel de-al doilea, un NULL drept corespondent al DataFact din primul SELECT. Clauza ORDER BY trebuie adăugată la finalul ultimului SELECT.

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	02-09-2007	110907
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 1 SRL - SUBTOTAL	(null)	5021287
Client 2 SA	02-09-2007	127530
Client 2 SA - SUBTOTAL	(null)	127530
Client 5 SRL	01-09-2007	153442
Client 5 SRL - SUBTOTAL	(null)	153442
Client 7 SRL	10-09-2007	136849
Client 7 SRL - SUBTOTAL	(null)	136849

Figura 7.15. Subtotaluri la nivel de clienți

Dacă trucul funcționează în Oracle, PostgreSQL și, prin schimbarea TRUNC-ului în ROUND și a EXTRACT-ului în DATEPART, și în MS SQL Server, în DB2 avem o problemă. Interogarea următoare:

```
SELECT DenCl AS DenumireClient, DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl=f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl, DataFact
                UNION
SELECT DenCl || ' - SUBTOTAL', NULL,
        TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0)
FROM clienti c
        INNER JOIN facturi f ON c.CodCl=f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl ORDER BY 1,2
```

se soldează cu eroarea din figura 7.16, eroare cauzată, aparent, de folosirea inadecvată a NULL-ului.

```
SELECT DenCl || ' - SUBTOTAL', NULL, TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0)
f.NrFact INNER JOIN produse p ON lf.CodPr=p.CodPr WHERE YEAR(DataFact) = 2007 AND MON
SQL0206N "NULL" is not valid in the context where it is used. SQLSTATE=42703
```

Figura 7.16. Eroare DB2 la folosirea NULL-ului în a doua interogare (cea de subtotal)

Soluţia este mai simplă decât vă aşteptaţi: se înlocuieşte, doar, NULL cu *CAST (NULL AS DATE)*. În plus, se foloseşte clauza AS în ambele interogări, astfel încât antetul rezultatului este corect[10] – vezi figura 7.17:

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	01.09.2007	4.442.959,0000
Client 1 SRL	02.09.2007	110.907,0000
Client 1 SRL	10.09.2007	287.855,0000
Client 1 SRL	17.09.2007	179.565,0000
Client 1 SRL - SUBTOTAL		5.021.287,0000
Client 2 SA	02.09.2007	127.530,0000
Client 2 SA - SUBTOTAL		127.530,0000
Client 3 SRL	07.09.2007	5.819.668,0000
Client 3 SRL - SUBTOTAL		5.819.668,0000
Client 5 SRL	01.09.2007	153.442,0000
Client 5 SRL - SUBTOTAL		153.442,0000
Client 7 SRL	10.09.2007	136.849,0000
Client 7 SRL - SUBTOTAL		136.849,0000

Figura 7.17. Obţinerea subtotalurilor şi totalului general în DB2

```
SELECT DenCl AS DenumireClient,
        DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl, DataFact
                UNION
SELECT DenCl || ' - SUBTOTAL' AS DenumireClient,
        CAST (NULL AS DATE) Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl, CAST (NULL AS DATE)
ORDER BY 1,2
```

10. Reţinem, deci, ideea: când, în DB2, o interogare constă în mai multe SELECT-uri reunite, singurul mod de a obţine un antet dorit pentru fiecare coloană este să se folosească clauza AS în fiecare SELECT.

La cele două SELECT-uri mai adăugăm un al treilea pentru afişarea totalului general. Pentru rândul de total general trebuie să avem în vedere faptul că valoarea coloanei DenumireClient trebuie să fie ultima la ordonare. Este motivul pentru care la început vom insera un z mic (oricum, prin restricţiile declarate în capitolul 3, denumirea fiecărui client trebuie să înceapă cu o majusculă), urmat de trei asteriscuri (pur decorative). Un alt element de noutate ţine de afişarea subtotalului la începutul fiecărui client, ca în figura 7.18.

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	subtotal (toate zilele)	5021287
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	02-09-2007	110907
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 2 SA	subtotal (toate zilele)	127530
Client 2 SA	02-09-2007	127530
Client 5 SRL	subtotal (toate zilele)	153442
Client 5 SRL	01-09-2007	153442
Client 7 SRL	subtotal (toate zilele)	136849
Client 7 SRL	10-09-2007	136849
z*** TOTAL GENERAL		5439108

Figura 7.18. Subtotaluri la început de client şi un total general

Pentru această formă a rezultatului, interogarea anterioară se modifică şi în SELECT-ul dedicat rândurilor „curente" - DataFact se va concatena cu spaţii pentru a se putea afişa, pentru rândurile de subtotal, mesajul „ subtotal (toate zilele) ", mesaj care începe cu un spaţiu tocmai pentru a fi primul la ordonare:

```
SELECT DenCl AS DenumireClient, DataFact || '    ' AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
        UNION
SELECT DenCl, ' subtotal (toate zilele) ',
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
```

```
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
        UNION
SELECT 'z*** TOTAL GENERAL', ' ',
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)
FROM facturi f
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
ORDER BY 1,2
```

În DB2 trebuie să modificăm doar pe ici, pe colo interogarea precedentă (Oracle/PostgreSQL):

```
SELECT ...,
        CAST(DataFact AS CHAR(10)) || '    ' AS Data,
        ...
FROM ...
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl, DataFact
        UNION
SELECT DenCl AS DenumireClient,
        ' subtotal (toate zilele) ' AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM ...
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY DenCl
        UNION
SELECT 'z*** TOTAL GENERAL' AS DenumireClient,
        ' ' AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM ...
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
ORDER BY 1,2
```

Ei bine, simţul artistic ne roade rău de tot şi ne întrebăm dacă nu am putea delimita începutul datelor despre un client, subtotalul unui client şi totalul general ca în figura 7.19. Aveţi dreptate! Am luat-o razna rău de tot (unde mai puneţi că în loc de egaluri voiam să pun iniţial inimioare). Fraza SELECT (sintaxa Oracle) este sfâşietoare:

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	================	=========
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	02-09-2007	110907
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 1 SRL	s----------------------	s--------------
Client 1 SRL	subtotal (toate zilele)	5021287
Client 2 SA	================	=========
Client 2 SA	02-09-2007	127530
Client 2 SA	s----------------------	s--------------
Client 2 SA	subtotal (toate zilele)	127530
Client 5 SRL	================	=========
Client 5 SRL	01-09-2007	153442
Client 5 SRL	s----------------------	s--------------
Client 5 SRL	subtotal (toate zilele)	153442
Client 7 SRL	================	=========
Client 7 SRL	10-09-2007	136849
Client 7 SRL	s----------------------	s--------------
Client 7 SRL	subtotal (toate zilele)	136849
z===============	===============	=========
z*** TOTAL GENERAL	++++++++++++++	5439108

Figura 7.19. Briz-brizuri pe la subtotaluri

```
SELECT DenCl AS DenumireClient, – acesta este SELECT-ul pentru randurile normale
        ' ' || DataFact AS Data,
        LPAD(TRIM(TRAILING FROM
                CAST(TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS
        CHAR(14))
                    ), 14, ' ') AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT(YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
        UNION
SELECT DenCl , – acesta este SELECT-ul pentru subtotaluri
        'subtotal (toate zilele) ',
        LPAD(TRIM (TRAILING FROM
                CAST(TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS
        CHAR(14))
                    ), 14, ' ')
```

```
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
        UNION
SELECT DenCl , – acesta este SELECT-ul pentru randul de la inceputul unui client
        ' ================',
        ' =========='
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
        UNION
SELECT DenCl , – acesta este SELECT-ul pentru randul dinaintea unui subtotal
        ' s————————— ',
        ' s——————'
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                        AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
        UNION
SELECT 'z*** TOTAL GENERAL', – acesta este SELECT-ul pentru randul de total
general
        ' ++++++++++++++',
        LPAD(TRIM(TRAILING FROM
                CAST(TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS CHAR(14))
                        ), 14, ' ')
FROM facturi f
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
```

WHERE EXTRACT(YEAR FROM DataFact) = 2007

AND EXTRACT (MONTH FROM DataFact) = 9

UNION

SELECT 'z===============', – **SELECT-ul pentru randul dinaintea totalui general**

'===============', '=========='

FROM dual

ORDER BY 1,2

La drept vorbind, ceea ce impresionează/intimidează la aceasta interogare este dimensiunea, nu că ar îngloba prea multă inteligență. Acesta este motivul pentru care nici nu ne mai sinchisim să o convertim în sintaxa celorlalte trei SGBD-uri.

Care este factura cu cea mai mare valoare?

Această interogare este prea pretențioasă pentru nivelul de SQL la care ne situăm în prezent. Cu toate acestea, Oracle prezintă o facilitate grozavă prin care putem redacta un răspuns: includerea unei funcții în alta (rezultatul este prezentat în figura 7.20):

SELECT MAX(SUM(Cantitate * PretUnit * (1 + ProcTVA)))

AS "Valoarea maxima a unei facturi"

FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr

GROUP BY NrFact

Valoarea maxima a unei facturi
6021706,5

Figura 7.20. Valoarea maximă a unei facturi (Oracle)

Din păcate, majoritatea SGBD-urilor nu „suportă" această facilitate - vezi „cazul" PostgreSQL în figura 7.21.

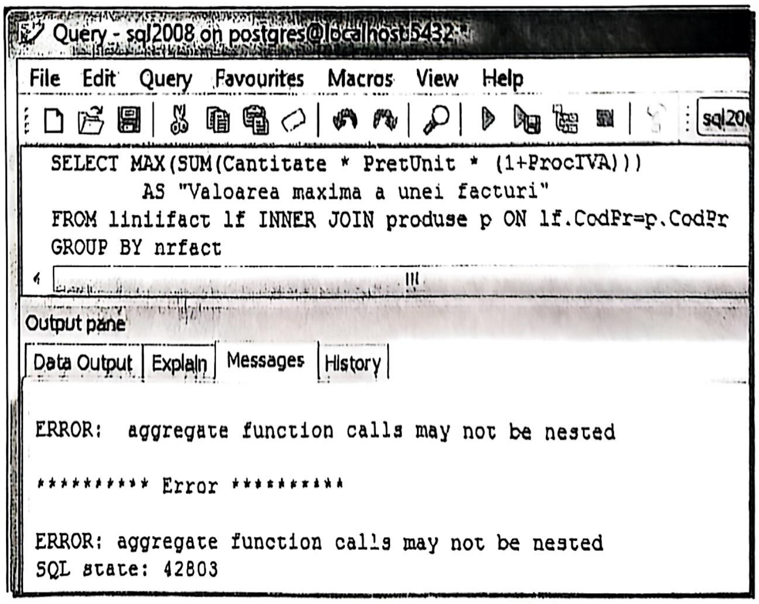


Figura 7.21. Imposibilitatea includerii unei funcții-agregat în altă funcție-agregat (în PostgreSQL)

De asemenea, cu opţiunile SQL de până acum, doar în Oracle putem afla şi numărul uneia (este posibil să fie mai multe) dintre facturile cu această valoare maximă:

```
SELECT MAX(
        TO_CHAR(
                SUM(Cantitate * PretUnit * (1 + ProcTVA)),
                '999999999999.99')
                || ', factura nr. ' || NrFact
        ) AS "Valoarea maxima a unei facturi"
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY nrfact
```

Valoarea maxima a unei facturi
6021706.50, factura nr. 1114

Figura 7.22. Valoarea maximă pentru o factură (şi numărul acesteia sau a uneia dintre ele, dacă sunt mai multe) – soluţie Oracle

7.3. Gruparea după funcţii şi expresii

A doua interogare din paragraful anterior - cea în care clauza GROUP BY are forma *GROUP BY DenPr || ' (' || UM || ')'* - sugerează că gruparea se poate realiza nu numai pentru atribute, ci şi după valorile anumitor expresii. Este ceea ce vrem să dezvoltăm în continuare.

Care este valoarea vânzărilor din fiecare zi a săptămânii?

Cu alte cuvinte, ne interesează o statistică prin care să putem compara cât s-a vândut lunea cu cât s-a vândut marţea ş.a.m.d., adică o situaţie în forma celei din figura 7.23.

ZI_SAPTAMINA	VINZARI_ZI_SAPT
luni	604269
marţi	25894871
miercuri	10847292
joi	745370
vineri	6041718
sâmbătă	4798376
duminică	238437

Figura 7.23. Vânzările pe zile ale săptămânii

Expresia de grupare are la bază discuția din paragrafele 6.3 și 6.5 (figura 6.37). În Oracle/PostgreSQL putem folosi varianta:

```
SELECT TO_CHAR(DataFact, 'day') AS Zi_Saptamana,
        TRUNC(
        SUM(Cantitate * PretUnit * (1 + ProcTVA))
        ,0) AS Vanzari_Zi_Sapt
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY TO_CHAR(DataFact, 'day')
```

Rezultatul este încurajator (vezi figura 7.24). Entuziasmul ne trece repede, întrucât realizăm că nu prea ne pricepem să ordonăm zilele. Cu ceea ce știm până în acest moment, cu siguranță nu suntem în stare să încropim o clauză ORDER BY corespunzătoare.

ZI_SAPTAMINA	VINZARI_ZI_SAPT
miercuri	10847292
duminica	6058105
marti	25894871
joi	745370
vineri	222050
sâmbata	4798376
luni	604269

Figura 7.24. Zilele săptămânii ordonate anapoda

Câteva speranțe de soluționare ar fi în PostgreSQL, întrucât, după cum am văzut în paragraful 6.3, în funcția EXTRACT poate fi folosit argumentul DOW - Day of the Week:

```
SELECT EXTRACT (DOW FROM DataFact) AS Zi_Saptamana,
        TRUNC(
        SUM(Cantitate * PretUnit * (1 + ProcTVA))
        ,0) AS Vanzari_Zi_Sapt
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY EXTRACT (DOW FROM DataFact)
ORDER BY EXTRACT (DOW FROM DataFact)
```

Ceea ce obținem la execuția acestei interogări - figura 7.25 - ne duce cu gândul la celebrele bancuri cu vestea bună și vestea rea. Vestea bună este că zilele sunt ordonate, iar vestea proastă este că nu dispunem de numele fiecărei zile, ci de numărul său - 0 pentru duminică, 1 pentru luni, ... și 6 pentru sâmbătă.

zi_saptamina double precis	vinzari_zi_sapt numeric
0	6058105
1	604269
2	25894871
3	10847292
4	745370
5	222050
6	4798376

Figura 7.25. Zilele săptămânii ordonate bine, dar identificate după număr în loc de nume

În afară de a ne manifesta nemulţumirea pentru inexistenţa unei clauze (de tip CDOW) care să furnizeze denumirea zilei din săptămână, suntem tentaţi să recurgem la o improvizaţie în care să facem gruparea după denumirea zilei din săptămână, păstrând ordonarea după numărul acesteia:

```
SELECT TO_CHAR(DataFact, 'day') AS Zi_Saptamana,
        TRUNC(
        SUM(Cantitate * PretUnit * (1 + ProcTVA))
        ,0) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY TO_CHAR(DataFact, 'day')
ORDER BY EXTRACT (DOW FROM DataFact)
```

Din păcate, PostgreSQL-ul tratează cu ostilitate improvizaţia noastră (vezi figura 7.26), aşa că nu ne rămâne decât să aşteptăm până la paragraful 8.5, unde vom folosi o structură CASE, care ne va face ordonarea după cum vom porunci.

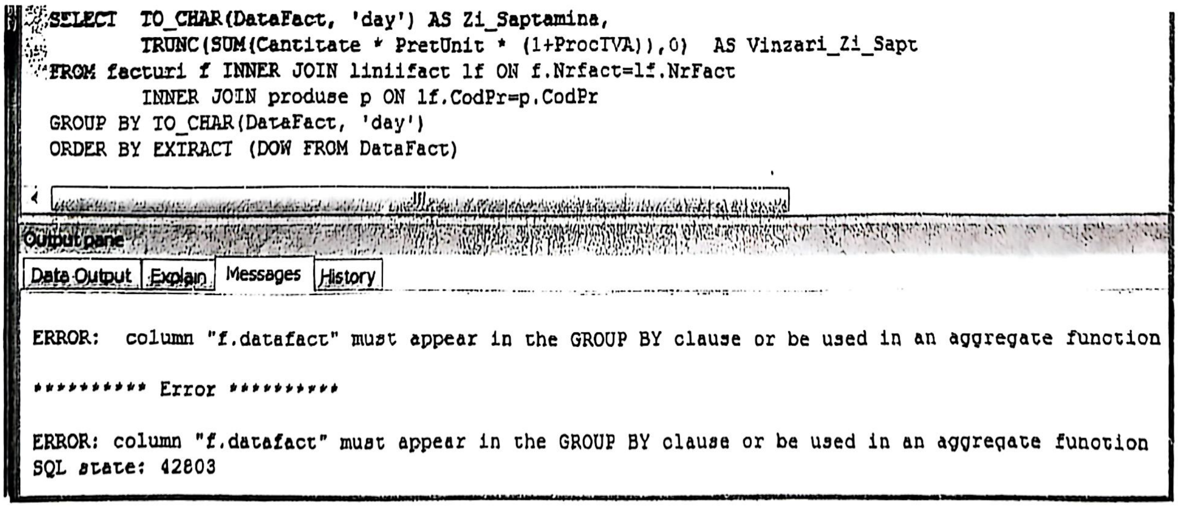


Figura 7.26. Improvizaţie ratată în PostgreSQL

Problema se prezintă similar şi în MS SQL Server, întrucât interogarea:

```
SELECT DATENAME(WEEKDAY, DataFact) AS Zi_Saptamana,
        ROUND(
```

```
        SUM(Cantitate * PretUnit * (1 + ProcTVA))
              ,0,1) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DATENAME(WEEKDAY, DataFact)
```

conduce la rezultatul din figura 7.27, iar încercarea de a modifica ordonarea după dispunerea firească a zilelor din săptămână:

Zi_Saptamina	Vinzari_Zi_Sapt
Friday	222050.0000
Monday	604269.0000
Saturday	4798376.0000
Sunday	6058105.0000
Thursday	745370.0000
Tuesday	25894871.0000
Wednesday	10847292.0000

Figura 7.27. Zilele săpămânii ordonate implicit în MS SQL Server

```
SELECT DATENAME(WEEKDAY, DataFact) AS Zi_Saptamana, ROUND(
        SUM(Cantitate * PretUnit * (1 + ProcTVA)),0,1) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DATENAME(WEEKDAY, DataFact)
ORDER BY DATEPART(WEEKDAY, DataFact)
```

are soarta din PostgreSQL. La fel stau lucrurile şi în DB2 (funcţiile folosite aici fiind DAYOFWEEK_ISO şi DAYNAME).

Să se afle numărul de produse vândute, pe următoarele intervale ale preţului unitar de vânzare: sub 1000 RON, între 1000 şi 1999, 2000 şi 2999 ş.a.m.d.

Chiar dacă enunţul nu este dintre cele mai fericite, genul acesta de problemă este unul întâlnit în practică, dificultatea ţinând de modul de grupare. Aici ne interesează pe câte linii, din toate facturile, preţul unitar a fost între 0 şi 999 RON, între 1000 şi 1999, între 2000 şi 2999 ş.a.m.d., ca în figura 7.28.

Pret unitar peste ... (RON)	Nr.aparitii
0	21
1000	26
6000	3
7000	6

Figura 7.28. Grupare pe intervale ale preţului unitar

Cheia soluţiei stă, după cum vă aşteptaţi, în expresia de grupare care este *TRUNC(PreţUnit,-3)*. Argumentul -3 aplicat funcţiei TRUNC asigură, după cum am văzut în paragraful 6.1, trunchierea la ordinul miilor:

```
SELECT TRUNC(PretUnit,-3) AS "Pret unitar peste ... (RON)",
        COUNT(*) AS "Nr. aparitii"
FROM liniifact
GROUP BY TRUNC(PretUnit,-3)
ORDER BY 1[11]
```

Singura nemulţumire pe care o putem avea este legată de modul de precizare a intervalelor, dar nu trebuie să fim prea tipicari.

Să se afle numărul de facturi întocmite în fiecare lună (calendaristică).

Deşi noi am populat tabelele FACTURI şi LINIIFACT cu câteva înregistrări doar pentru lunile august şi septembrie 2007, trebuie avut în vedere că, în practică, baza de date este folosită ani şi chiar decenii întregi, aşa că trebuie să asigurăm diferenţierea datelor din septembrie 2007, de cele din septembrie 2006, septembrie 2005 etc.

Pe scurt, avem cel puţin două soluţii de grupare. În prima, clauza GROUP BY va conţine două expresii/funcţii care vor extrage anul şi luna (interogarea următoare este redactată pe sintaxa Oracle/PostgreSQL):

```
SELECT EXTRACT (YEAR FROM DataFact) AS An,
        EXTRACT (MONTH FROM DataFact) AS Luna,
        COUNT(*) AS Nr_facturi
FROM facturi
GROUP BY EXTRACT (YEAR FROM DataFact),
        EXTRACT (MONTH FROM DataFact)
ORDER BY 1,2
```

A doua (tot Oracle/PostgreSQL) va folosi o funcţie de extragere a lunii din data facturii, care poate fi TO_CHAR(DataFact, 'YYYY-MM'):

```
SELECT TO_CHAR(DataFact, 'YYYY-MM') AS An_Luna, COUNT(*) AS Nr_facturi
FROM facturi
GROUP BY TO_CHAR(DataFact, 'YYYY-MM')
ORDER BY 1,2
```

sau TRUNC(DataFact,'MONTH') în Oracle, respectiv DATE_TRUNC('MONTH', DataFact) în PostgreSQL:

```
SELECT TRUNC(DataFact,'MONTH') AS "Dupa...", COUNT(*) AS Nr_facturi
FROM facturi
GROUP BY TRUNC(DataFact,'MONTH')
ORDER BY 1
```

11. Sintaxă DB2, Oracle şi PostgreSQL. Varianta corectă SQL Server este: SELECT ROUND(PretUnit,-3,1) AS "Pret unitar peste ... (RON)", COUNT(*) AS "Nr. aparitii" FROM liniifact GROUP BY ROUND (PreţUnit,-3,1) ORDER BY 1.

Diferențele dintre cele trei variante sunt doar de fațadă (vezi figura 7.29).

AN	LUNA	NR_FACTURI
2007	8	22
2007	9	7
2007	10	1

AN_LUNA	NR_FACTURI
2007-08	22
2007-09	7
2007-10	1

Dupa	NR_FACTURI
01-08-2007	22
01-09-2007	7
01-10-2007	1

Figura 7.29. Trei soluții la o problemă simplă de grupare pe luni calendaristice

O variantă funcțională în MS SQL Server este:

```
SELECT DATEPART(YEAR, DataFact) AS An,
        DATEPART(MONTH, DataFact) AS Luna, COUNT(*) AS Nr_facturi
FROM facturi
GROUP BY DATEPART(YEAR, DataFact), DATEPART(MONTH, DataFact)
ORDER BY 1,2
```

iar în DB2:

```
SELECT YEAR(DataFact) AS An, MONTH(DataFact) AS Luna,
        COUNT(*) AS Nr_facturi
FROM facturi
GROUP BY YEAR(DataFact), MONTH(DataFact)
ORDER BY 1,2
```

Să se calculeze vânzările săptămânale.

Neavând prea multe înregistrări în tabele, am renunțat la filtrarea pe luni, trimestre, semestre sau ani. Pentru a rezolva problema, trebuie să găsim o soluție prin care să determinăm începutul unei săptămâni (după aceea, aflarea sfârşitului săptămânii e floare la ureche). Funcția TRUNC, care, încet-încet, ne devine chiar simpatică, ne pune la dispoziția opțiunile 'D', 'DAY' şi 'DY' ce furnizează data de început a unei săptămâni (însă trebuie corectată cu o zi, întrucât săptămâna începe în Oracle cu duminica):

```
SELECT TRUNC(DataFact, 'd') + INTERVAL '1' DAY AS "De la...",
        TRUNC(DataFact, 'd') + INTERVAL '7' DAY AS "Pana la...",
        TRUNC(
                    SUM(Cantitate * PretUnit * (1 + ProcTVA))
              ,0) AS Vanzari_Sapt
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY TRUNC(DataFact, 'd') + INTERVAL '1' DAY,
        TRUNC(DataFact, 'd') + INTERVAL '7' DAY
ORDER BY 1
```

Soluţia funcţionează în Oracle (vezi în figura 7.30), însă în PostgreSQL trebuie să folosim, ca şi mai sus, funcţia DATE_TRUNC (fără a mai avea nevoie de corecţia de o zi):

De la...	Pana la...	VINZARI_SAPT
31-07-2007	06-08-2007	11301510
07-08-2007	13-08-2007	10610000
14-08-2007	20-08-2007	5439108
21-08-2007	27-08-2007	10560939
28-08-2007	03-09-2007	4834839
11-09-2007	17-09-2007	424704
18-09-2007	24-09-2007	179565
02-10-2007	08-10-2007	5819668

Figura 7.30. Vânzări săptămânale

```
SELECT DATE_TRUNC('week',DataFact) AS "De la...",
        DATE_TRUNC('week',DataFact) + INTERVAL '6 DAYS'
                    AS "Pana la...",
        TRUNC(
                SUM(Cantitate * PretUnit * (1 + ProcTVA))
                ,0) AS Vanzari_Sapt
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DATE_TRUNC('week', DataFact), DATE_TRUNC('week', DataFact) +
                INTERVAL '6 DAYS'
ORDER BY 1
```

Pentru a furniza răspunsul la această problemă, în MS SQL SERVER trebuie să ţinem cont că nu beneficiem de elegantele funcţii TRUNC/DATE_TRUNC, ci doar de DATEPART şi ADDDATE. Astfel, putem afla săptămâna anului în care pică fiecare factură, însă aflarea datei de început şi de sfârşit a fiecărei săptămâni necesită adunarea la data de 1 ianuarie (dată obţinută prin concatenarea anului facturii cu '-01-01' (1 ianuarie) şi transformarea şirului în dată calendaristică) a numărului săptămânii determinat cu *DATEPART(WEEK, DataFact)*. Datorită faptului că, prin adunarea la prima zi din an a unui număr de săptămâni, se determină prima zi a săptămânii viitoare (din cauza setării curente a primei zile din săptămână), „poziţionarea" zilei de început şi sfârşit a fiecărei săptămâni necesită un „deplasament": de 7 zile, respectiv o zi „înainte" (vezi figura 7.31):

```
SELECT NrFact, DataFact, DATEPART(WEEK, DataFact) AS Saptamana_Nr,
        CAST(STR(YEAR(DataFact),4) + '-01-01' AS SMALLDATETIME)
                AS Zi_Inceput_An,
        DATEADD(WEEK, DATEPART(WEEK, DataFact),
```

```
CAST(STR(YEAR(DataFact),4)+'-01-01' AS SMALLDATETIME)) - 7
        AS Prima_zi_a_sapt,
DATEADD(WEEK, DATEPART(WEEK, DataFact),
CAST(STR(YEAR(DataFact),4)+'-01-01' AS SMALLDATETIME)) - 1
        AS Ultima_zi_a_sapt
FROM facturi f
```

NrFact	DataFact	Saptamina_Nr	Zi_Inceput_An	Prima_zi_a_sapt	Ultima_zi_a_sapt
1111	2007-08-01 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1112	2007-08-01 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1113	2007-08-01 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1114	2007-08-01 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1115	2007-08-02 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1116	2007-08-02 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1117	2007-08-03 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1118	2007-08-04 00:00:00	31	2007-01-01 00:00:00	2007-07-30 00:00:00	2007-08-05 00:00:00
1119	2007-08-07 00:00:00	32	2007-01-01 00:00:00	2007-08-06 00:00:00	2007-08-12 00:00:00
1120	2007-08-07 00:00:00	32	2007-01-01 00:00:00	2007-08-06 00:00:00	2007-08-12 00:00:00
1121	2007-08-07 00:00:00	32	2007-01-01 00:00:00	2007-08-06 00:00:00	2007-08-12 00:00:00
1122	2007-08-07 00:00:00	32	2007-01-01 00:00:00	2007-08-06 00:00:00	2007-08-12 00:00:00

Figura 7.31. „Manevre săptămânale" în SQL Server

Pe baza acestor aranjamente obținem aceleași vânzări săptămânale ca în figura 7.30, folosind interogarea:

```
SELECT DATEPART(WEEK, DataFact) AS Saptamana_Nr,
       DATEADD(WEEK, DATEPART(WEEK, DataFact),
               CAST(STR(YEAR(DataFact),4) + '-01-01' AS SMALLDATETIME)) - 7
                  AS Prima_zi_a_sapt,
       DATEADD(WEEK, DATEPART(WEEK, DataFact),
               CAST(STR(YEAR(DataFact),4) + '-01-01' AS SMALLDATETIME)) - 1
                  AS Ultima_zi_a_sapt,
       ROUND(
               SUM(Cantitate * PretUnit * (1 + ProcTVA))
               ,0,1) AS Vanzari_Sapt
 FROM facturi f
       INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
       INNER JOIN produse p ON lf.CodPr = p.CodPr
 GROUP BY DATEPART(WEEK, DataFact),
       DATEADD(WEEK, DATEPART(WEEK, DataFact),
               CAST(STR(YEAR(DataFact),4) + '-01-01' AS SMALLDATETIME)) - 7,
       DATEADD(WEEK, DATEPART(WEEK, DataFact),
               CAST(STR(YEAR(DataFact),4) + '-01-01' AS SMALLDATETIME)) - 1
```

Sintaxa DB2 de obținere a rezultatului din figura 7.30 este:

```
SELECT DATE(CAST(YEAR(DataFact) AS CHAR(4)) || '-01-01') +
                ((WEEK_ISO(DataFact) - 1) * 7) DAYS AS "De la...",
        DATE(CAST(YEAR(DataFact) AS CHAR(4)) || '-01-01') +
                ((WEEK_ISO(DataFact)) * 7) DAYS AS "Pana la...",
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA))
                ,0) AS Vanzari_Sapt
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY YEAR(DataFact),WEEK_ISO(DataFact),
        DATE(CAST(YEAR(DataFact) AS CHAR(4)) || '-01-01') +
                WEEK_ISO(DataFact) DAYS,
        DATE(CAST(YEAR(DataFact) AS CHAR(4)) || '-01-01') +
                ((WEEK_ISO(DataFact)) * 7) DAYS
```

7.4. Clauza HAVING

Cea mai simplă definiție ar suna cam așa: clauza HAVING este WHERE-ul ce operează la nivel de grupuri (un fel de a zice că „HAVING este pentru noi... un fel de WHERE doi"). Dacă WHERE acționează la nivel de *tuplu*, selectând acele linii care îndeplinesc o condiție specificată, HAVING permite specificarea unor condiții de selecție care se aplică *grupurilor* de linii create prin clauza GROUP BY. Formatul general al unei interogări ce conține clauza HAVING este:

```
SELECT coloana 1, coloana 2, ... , coloana m
FROM tabela
WHERE predicat-pentru-tupluri
GROUP BY coloană-de-grupare
HAVING predicat-pentru-grupuri
```

Care sunt zilele în care s-au întocmit cel puțin trei facturi?

În al doilea exemplu din paragraful anterior (figurile 7.2 și 7.3) am văzut mecanismul de grupare care ne permitea să aflăm numărul de facturi întocmit în fiecare zi cu vânzări. La pașii derulați într-o operațiune de grupare se mai adaugă unul prin care, dintre toate grupurile, se extrag în rezultatul final doar cele care îndeplinesc condiția specificată în clauza HAVING (vezi figura 7.33):

```
SELECT DataFact AS "Zi", COUNT(*) AS "Numar facturi"
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) >= 3
```

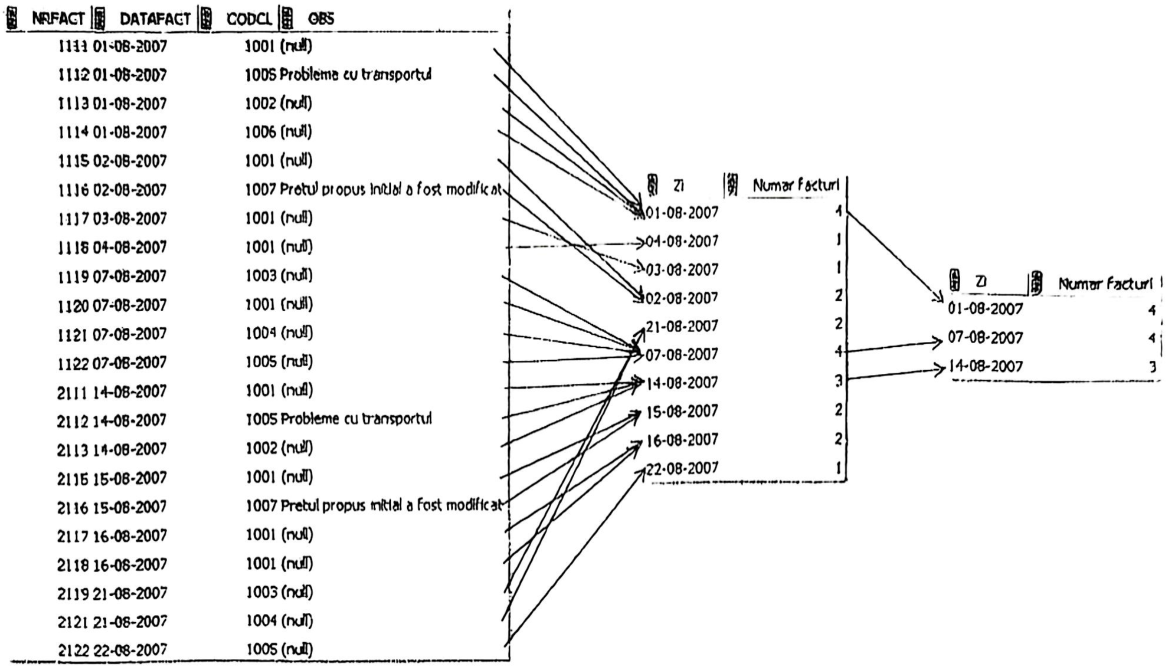


Figura 7.33. Logica execuţiei clauzelor GROUP BY şi HAVING

În lipsa clauzei ORDER BY nu putem fi siguri de ordinea în care vor apărea rândurile în rezultat, cu atât mai puţin ordinea de constituire a grupurilor.

Care sunt zilele din luna septembrie 2007 în care valoarea (cu TVA a) vânzărilor este mai mare decât 300000 RON?

Rezultatul interogării care urmează (valabilă în Oracle şi PostgreSQL) ar trebui să conţină una sau mai multe linii dintre cele din figura 7.4:

```
SELECT DataFact, TRUNC(
        SUM(Cantitate * PretUnit * (1 + ProcTVA))
        ,0) AS ValTotala
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT(YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DataFact
HAVING TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) > 300000
ORDER BY 1
```

Aşa şi este, dacă examinăm figura 7.34.

DATAFACT	VALTOTALA
01-09-2007	4596401
10-09-2007	424704

Figura 7.34. Zilele cu vânzări de peste 300000 RON

Pentru care clienți și în ce zile din luna septembrie 2007 vânzările au fost mai mari de 150000 RON?

Faptul că avem o grupare după două atribute (DenCl și DataFact) nu influențează cu nimic clauza HAVING. Ne oprim doar la sintaxa Oracle/PostgreSQL a interogării la execuția căreia obținem figura 7.35.

```
SELECT DenCl AS DenumireClient,
        DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT(YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
HAVING TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) > 150000
ORDER BY 1,2
```

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 5 SRL	01-09-2007	153442

Figura 7.35. Clienții și zilele din septembrie 2007 pentru care vânzările au depășit pragul de 150000

Unul dintre reproșurile aduse SQL-ului din „tabăra" relaționalului este că, pentru aceeași problemă există o sumedenie de soluții, ceea ce ar genera confuzii (în termeni elevați, se vorbește despre *supraîncărcare semantică*). Eu, unul, nu împărtășesc această supărare, ba, din contra, sunt mulțumit că am de ales. Să luăm un exemplu simplu pe care să formulăm două soluții echivalente, din punctul de vedere al rezultatului.

Să se afișeze vânzările zilnice din luna septembrie 2007 corespunzătoare Clientului 1 SRL, rezultatul având forma propusă în figura 7.36.

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	01-09-2007	4442959
Client 1 SRL	02-09-2007	110907
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565

Figura 7.36. Vânzările zilnice către Client 1 SRL pe luna septembrie 2007

Ne oprim tot la sintaxa PostgreSQL/Oracle.

Soluţia 1 – filtrarea clientului în clauza WHERE şi grupare după *(DenCl, DataFact)*:

```
SELECT DenCl AS DenumireClient,
        DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH
        FROM DataFact) = 9 AND DenCl = 'Client 1 SRL'
GROUP BY DenCl, DataFact
ORDER BY 1,2
```

Soluţia 2 – grupare tot după *(DenCl, DataFact)*, însă filtrarea clientului se face în clauza HAVING:

```
SELECT DenCl AS DenumireClient,
        DataFact AS Data,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DataFact
HAVING DenCl = 'Client 1 SRL'
ORDER BY 1,2
```

Chiar dacă cele două soluţii sunt echivalente ca rezultat, consumul lor de resurse este unul diferit şi putem spune că, într-o aplicaţie „aglomerată" (de mare complexitate şi cu mulţi utilizatori ce lucrează, direct sau indirect, cu baza de date), alegerea uneia în detrimentul celeilalte nu este întâmplătoare.

În ce zile s-au emis mai multe facturi decât pe 2 august 2007?

Problema este ceva mai exotică, iar soluţia este una pe măsură:

```
SELECT F1.DataFact AS Zi1, COUNT(DISTINCT F1.NrFact) AS Nr_Facturilor1,
        F2.DataFact AS Zi2, COUNT(DISTINCT F2.NrFact) AS Nr_Facturilor2
FROM facturi F1, facturi F2
WHERE F2.DataFact = DATE'2007-08-02'
GROUP BY F1.DataFact, F2.DataFact
HAVING COUNT(DISTINCT F1.NrFact) > COUNT(DISTINCT F2.NrFact)
```

	F1.NrFact	F1.DataFact	F2.NrFact	F2.DataFact
Grup 1	1111	01-08-2007	1115	02-08-2007
	1111	01-08-2007	1116	02-08-2007
	1112	01-08-2007	1115	02-08-2007
	1112	01-08-2007	1116	02-08-2007
	1113	01-08-2007	1115	02-08-2007
	1113	01-08-2007	1116	02-08-2007
	1114	01-08-2007	1115	02-08-2007
	1114	01-08-2007	1116	02-08-2007
Grup 2	1115	02-08-2007	1115	02-08-2007
	1115	02-08-2007	1116	02-08-2007
	1116	02-08-2007	1115	02-08-2007
	1116	02-08-2007	1116	02-08-2007
Grup 3	1117	03-08-2007	1115	02-08-2007
	1117	03-08-2007	1116	02-08-2007
Grup 4	1118	04-08-2007	1115	02-08-2007
	1118	04-08-2007	1116	02-08-2007
...	[illegible]	[illegible]	[illegible]	[illegible]
	...			

Figura 7.37. Produs cartezian a două instanțe ale tabelei FACTURI, urmat de selectarea, în instanța a doua, numai a facturilor de pe 2 august (parțial)

Să zăbovim asupra logicii acestei interogări[13]. Mai întâi se efectuează produsul cartezian pentru două instanțe (F1 şi F2) ale tabelei FACTURI, eliminându-se pentru F2 liniile în care data este alta decât 2 august 2007, adică:

```
SELECT F1.NrFact AS "F1.NrFact", F1.DataFact AS "F1.DataFact",
        F2.NrFact AS "F2.NrFact", F2.DataFact AS "F2.DataFact"
FROM facturi F1, facturi F2
WHERE F2.DataFact = DATE'2007-08-02'
ORDER BY F1.DataFact, F1.NrFact, F2.DataFact, F2.NrFact
```

O parte din rezultatul acestui SELECT „intermediar" este afişată în figura 7.37. Se constituie apoi grupuri pentru fiecare combinație de valori *(F1.DataFact, F2.DataFact)*. F2.DataFact are aceeaşi valoare, *02-08-2007*, prin urmare, grupurile se constituie în funcție de F1.DataFact. Figura conține liniile primelor patru grupuri.

13. În DB2 şi SQL Server trebuie eliminat cuvântul DATE din interogare, ceea ce este valabil şi pentru următoarele trei interogări.

ZI1	NR_FACTURILOR1	ZI2	NR_FACTURILOR2
01-08-2007	4	02-08-2007	2
02-08-2007	2	02-08-2007	2
03-08-2007	1	02-08-2007	2
04-08-2007	1	02-08-2007	2
07-08-2007	4	02-08-2007	2
14-08-2007	3	02-08-2007	2
15-08-2007	2	02-08-2007	2
16-08-2007	2	02-08-2007	2
21-08-2007	2	02-08-2007	2
22-08-2007	1	02-08-2007	2
01-09-2007	2	02-08-2007	2
02-09-2007	2	02-08-2007	2
10-09-2007	2	02-08-2007	2
17-09-2007	1	02-08-2007	2
07-10-2007	1	02-08-2007	2

Figura 7.38. Gruparea liniilor din figura 7.37 după cele două instanţe ale atributului DataFact

În continuare, pe liniile rezultatului din figura 7.38 se aplică gruparea:

```
SELECT F1.DataFact AS Zi1, COUNT(DISTINCT F1.NrFact) AS Nr_Facturilor1,
       F2.DataFact AS Zi2, COUNT(DISTINCT F2.NrFact) AS Nr_Facturilor2
FROM facturi F1, facturi F2
WHERE F2.DataFact = DATE'2007-08-02'
GROUP BY F1.DataFact, F2.DataFact
ORDER BY 1
```

După cum se observă în figura 7.38, rezultatul are două jumătăţi necorelate în vreun fel. Cea din stânga conţine numărul de facturi pentru fiecare zi cu vânzări (pentru ca funcţia COUNT să calculeze corect numărul facturilor dintr-o zi, este necesară clauza DISTINCT), iar în dreapta se repetă linia corespunzătoare datei-etalon, 2 august 2007.

Acum nu mai rămâne decât să aplicăm condiţia de filtrare la nivel de grup şi să extragem în rezultat doar coloanele din prima „instanţă" din FACTURI:

```
SELECT F1.DataFact AS Zi1, COUNT(DISTINCT F1.NrFact) AS Nr_Facturilor1
FROM facturi F1, facturi F2
WHERE F2.DataFact = DATE'2007-08-02'
GROUP BY F1.DataFact, F2.DataFact
HAVING COUNT(DISTINCT F1.NrFact) > COUNT(DISTINCT F2.NrFact)
ORDER BY 1
```

Pentru conţinutul tabelei FACTURI din acest moment, există trei zile cu mai mult de două facturi (doi fiind numărul facturilor emise pe 2 august 2007), 1, 7 şi 14 august 2007 (vezi figura 7.39).

ZI1	NR_FACTURILOR1
01-08-2007	4
07-08-2007	4
14-08-2007	3

Figura 7.39. Zilele cu facturi mai multe decât în 2 august 2007

Cât mai este de încasat pentru fiecare factură?

Aşa cum reiese din figura 7.40, soluţia următoare (Oracle/PostgreSQL) oferă un rezultat incomplet. Lipsesc facturile care nu au nicio tranşă de încasare. Pentru rezolvarea cazului avem nevoie de clauza HAVING, dar şi de joncţiunea externă, după cum vom vedea în capitolul următor.

```
SELECT f.NrFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT i.CodInc)) AS Facturat,
        TRUNC(SUM(Transa)/MAX(lf.Linie)) AS Incasat,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT i.CodInc)) -
                        TRUNC(SUM(Transa)/MAX(lf.Linie)) AS Rest_de_incasat
FROM clienti c
        INNER JOIN facturi f ON c.CodCl=f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
        INNER JOIN incasfact i ON f.NrFact=i.NrFact
GROUP BY f.NrFact
```

NRFACT	FACTURAT	INCASAT	REST_DE_INCASAT
1111	4346037	53996	4292041
1112	125516	125516	0
1113	106275	106275	0
1117	222050	23204	198846
1118	201975	201975	0
1120	97664	7315	90349

Figura 7.40. Tentativă eşuată de a afla restul de încasat pentru fiecare factură

Care sunt facturile încasate măcar în proporţie de 10% ?

Gruparea o facem pe facturi (NrFact), iar extragerea celor care ne interesează se realizează prin clauza HAVING:

```
SELECT f.NrFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT i.CodInc)) AS Facturat,
```

```
        TRUNC(SUM(Transa)/MAX(lf.Linie)) AS Incasat,
                TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                        COUNT(DISTINCT i.CodInc)) -
                        TRUNC(SUM(Transa) / MAX(lf.Linie)) AS Rest_de_incasat
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN incasfact i ON f.NrFact = i.NrFact
GROUP BY f.NrFact
HAVING TRUNC(SUM(Transa)/MAX(lf.Linie)) >=
        (TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT i.CodInc))) * 0.10
```

Sintaxa este Oracle/PostgreSQL. După se observă în figura 7.41, sunt patru facturi în această situație, dintre care trei sunt încasate în totalitate.

NRFACT	FACTURAT	INCASAT	REST_DE_INCASAT
1112	125516	125516	0
1113	106275	106275	0
1117	222050	23204	198846
1118	201975	201975	0

Figura 7.41. Facturile încasate în proporție de minimum 10%

7.5. Diviziunea relațională

Câteva situații ce reclamă diviziunea relațională pot fi soluționate elegant cu ajutorul clauzelor GROUP BY şi HAVING. În exemplul următor (exemplul 24 din paragraful 4.4.7) este vorba despre o intersecție „simulată" printr-un mecanism apropiat de diviziune.

În ce zile s-au vândut şi produsul cu denumirea „Produs 1", şi cel cu denumirea „Produs 2"?

```
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
WHERE DenPr IN ('Produs 1', 'Produs 2')
GROUP BY DataFact
HAVING COUNT(DISTINCT liniifact.CodPr) = 2
ORDER BY DataFact
```

Sintaxa este operațională pentru toate cele patru servere BD. Prin joncțiunea celor trei tabele și selectarea liniilor care se referă strict la unul dintre cele două produse, obținem o coresрndență zile-produse ca în stânga figurii 7.42.

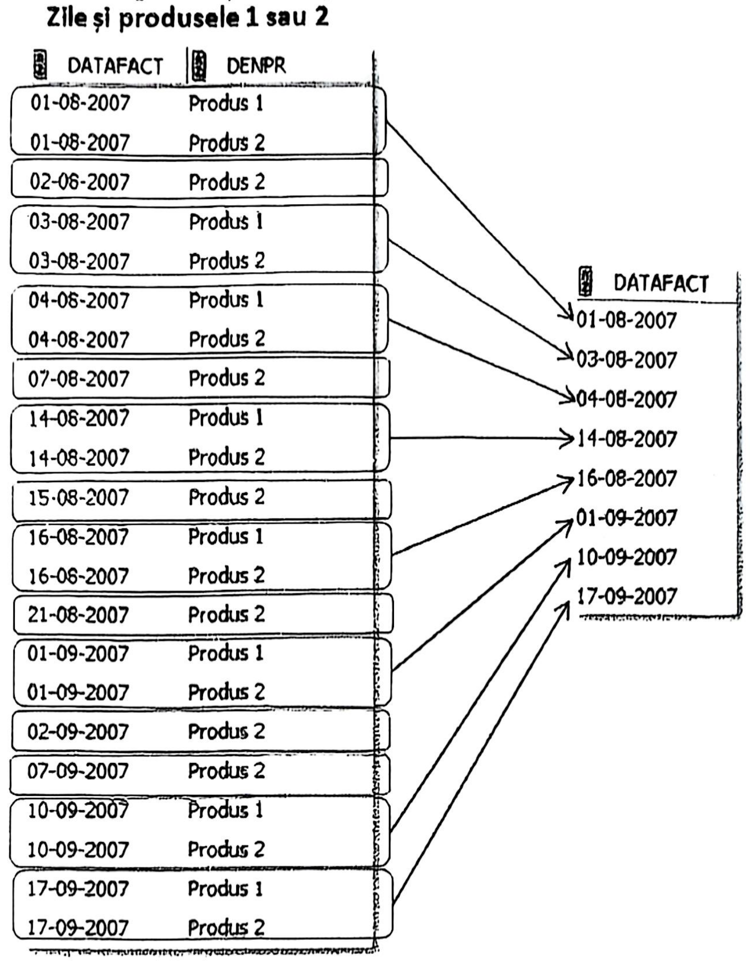


Figura 7.42. Zilele în care s-au vândut și Produs 1, și Produs 2

Un grup se asociază unei zile de vânzări în care s-a vândut fie numai Produs 1, fie numai Produs 2, fie ambele (singure sau în combinație cu alte produse). Pentru fiecare grup se numără câte produse, din Produs 1 și Produs 2, au fost facturate. Funcția COUNT() din clauza HAVING poate „întoarce" maxim valoarea 2, caz în care data respectivă se încadrează în zilele căutate.

Celelalte probleme de diviziune relațională reclamă includerea în predicatul clauzei HAVING a uneia sau mai multor consultări (subconsultări). Dar despre această facilitate vom vorbi în capitolul 9 (paragraful 9.3).

Capitolul 8
NULL-ităţi, joncţiuni externe şi structuri alternative

Alăturarea celor trei subiecte în acest capitol are la bază o doză substanţială de relativism. Totuşi, de joncţiuni externe nu putem sufla o vorbă dacă am fi ignoranţi în materie de NULL, iar structurile de tip CASE, ca funcţionalitate, pot fi legate destul de natural de celelalte două teme.

8.1. NULL-i în tabele

Valoarea NULL a fost introdusă în capitolul 2, la explicarea noţiunilor modelului relaţional, ca posibilitate de reprezentate a informaţiilor... inexistente sau inaplicabile. Raportul Interim 75-02-09 înaintat ANSI X3 (SPARC Study Group 1975) a delimitat 14 tipuri de date incomplete ce ar putea apărea ca rezultate ale unor operaţii sau valori ale atributelor, printre care: depăşiri ale capacităţii de stocare, diviziune la zero, trunchierea şirurilor de caractere, ridicarea lui zero la puterea zero şi alte erori computaţionale, precum şi valori necunoscute.

8.1.1. Valori nule în bazele de date TRIAJ şi VÂNZĂRI

O parte dintre problemele nulităţii atributelor din tabelele celor două baze de date au fost discutate pe parcursul capitolelor 2 şi 3. Tipologia NULL-ităţilor în BD TRIAJ este destul de simplă. Tabela TRIAJ este singura în care apar, deocamdată, valori NULL (vezi listingurile de populare din capitolul 3). Astfel, în primul caz din triaj (prima linie a tabelei TRIAJ) care a fost recepţionat/investigat pe 3 ianuarie 2008, la ora 7:18, valoarea atributului Tratament_Imediat este NULL, ceea ce înseamnă că nu i-a fost administrat, pe loc, niciun medicament sau vreo procedură. Pe a treia linie, cazul recepţionat tot pe 3 ianuarie 2008, dar la ora 12:45, valoarea atributului Secţie_Destinaţie este NULL, ceea ce ar putea însemna că starea pacientului nu era îngrijorătoare; pacientul şi-a revenit după o scurtă medicaţie; nemaiexistând alte simptome, acesta a fost externat.

În schimb, pentru tabelele bazei de date VÂNZĂRI, aveam trei situaţii majore:

- valoare necunoscută, cum ar fi cea a adresei Clientului 2 SA;
- valoare inaplicabilă, cum ar valoarea unui eventual atributului Culoare în tabela PRODUSE, pentru sortimente precum Vodcă, Sare iodată, Cafea etc.;

- valoare inexistentă – cum ar fi Ioan Vasile (tabela PERSOANE) care nu şi-a deschis, încă niciun cont de poştă electronică (e-mail).

De multe ori, nici nu ştim la care dintre aceste situaţii se referă (meta)valoarea NULL. Practic, nu ştim exact dacă Ioan Vasile este tehnofob şi nu vrea să audă de e-mail sau dacă şi-a deschis un cont între timp, dar n-am aflat noi (şi soţia).

Cum pot apărea valori nule într-o tabelă? Simplu: fie la inserarea unei linii într-o tabelă (INSERT), fie la modificarea valorii unui atribut pe o linie dintr-o tabelă (UPDATE). Bun, ar mai fi o modalitate – prin proceduri stocate, dar nu ne interesează lucrurile acestea acum (adică ne interesează, dar nu ne pricepem). Din capitolul 3 ştim că este posibil ca la inserare să se specifice numai valorile anumitor atribute:

INSERT INTO persoane (CNP, Nume, Prenume, Sex, CodPost, TelAcasa)
VALUES ('CNP10', 'Procopiu', 'Ivan', 'B' , '700505', NULL);

Clauza VALUES conţine numai valorile atributelor enumerate după numele tabelei, adică CNP, Nume, Prenume, Sex, CodPoşt şi TelAcasă. Cum pentru atributele Adresă, TelBirou, TelMobil şi Email nu am definit în capitolul 3 valori implicite (DEFAULT), în tabelă, pentru linia proaspăt inserată valorile acestora vor fi NULL (vezi figura 8.1). Dar şi clauza VALUES are un NULL, aşa încât şi atributul TelAcasă va avea aceeaşi soartă.

```
INSERT INTO persoane (CNP, Nume, Prenume, Sex, CodPost, TelAcasa)
        VALUES ('CNP10', 'Procopiu', 'Ivan', 'B' , '700505', NULL) ;

SELECT * FROM persoane
```

Results | Messages

	CNP	Nume	Prenume	Adresa	Sex	CodPost	TelAcasa	TelBirou	TelMobil	Email
1	CNP1	Ioan	Vasile	I.L.Caragiale, 22	B	700505	123456	987654	094222222	NULL
2	CNP10	Procopiu	Ivan	NULL	B	700505	NULL	NULL	NULL	NULL
3	CNP2	Vasile	Ion	NULL	B	700505	234567	876543	094222223	Ion@a.ro
4	CNP3	Popovici	Ioana	V.Micle, Bl.I, Sc.B,Ap.2	F	701150	345678	NULL	094222224	NULL
5	CNP4	Lazar	Caraion	M.Eminescu, 42	B	706500	456789	NULL	094222225	NULL
6	CNP5	Iurea	Simion	I.Creanga, 44 bis	B	706500	567890	543210	NULL	NULL
7	CNP6	Vasc	Simona	M.Eminescu, 13	F	701150	NULL	432109	094222227	NULL
8	CNP7	Popa	Ioanid	I.Ion, Bl.H2, Sc.C, Ap.45	B	701900	789012	321098	NULL	NULL
9	CNP8	Bogacs	Ildiko	I.V.Viteazu, 67	F	705550	890123	210987	094222229	NULL
10	CNP9	Ioan	Vasilica	Garii, Bl.B4, Sc.A, Ap.1	F	701900	901234	109876	094222230	NULL

Figura 8.1. Valorile din baza de date pentru linia inserată

La modificare, lucrurile sunt relativ simple. Dacă aflăm că numărul de mobil al Ioanei Popovici (CNP3) nu mai este actual, pentru a evita deranjurile (telefonice), până la aflarea numărului corect putem să i-l „nulizăm" (un pic):

UPDATE persoane SET TelMobil = NULL WHERE CNP = 'CNP3'

În capitolul 3 am văzut că o parte dintre atribute – cele ale căror valori considerate obligatorii – pot scăpa de valorile NULL şi, implicit, de eventuale necazuri pricinuite de acestea, cu ajutorul restricţiei NOT NULL declarate fie la

crearea tabelei (CREATE TABLE), fie ulterior (ALTER TABLE). Dacă în urma unei inserări/modificări se încearcă atribuirea valorii NULL vreunui atribut declarat NOT NULL (de exemplu, Județ din tabela JUDEȚE) :

UPDATE judete

SET Judet = NULL

WHERE Jud = 'IS'

SGBD-ul (Oracle, în cazul nostru) va returna o eroare ce blochează inserarea/modificarea respectivă (vezi figura 8.2).

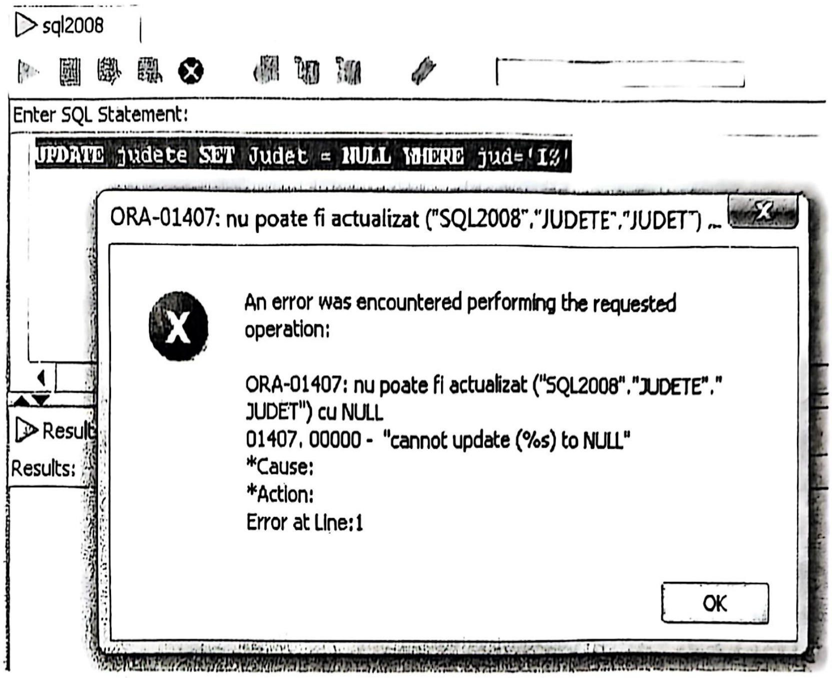


Figura 8.2. Modificare ce violează restricția de nenulitate

În paragraful 5.5 am introdus un mic ghimpe NULL-ist, întrucât, dacă introducem în lista operatorului NOT IN o valoare NULL, rezultatul nu va conține nicio linie. Probabil că ați rămas cu ideea că, fără negație, operatorul IN funcționează uns chiar dacă în listă strecurăm discret un NULL. Ei, bine, nu-i chiar așa! În tabela CLIENȚI, pentru cei care au codurile 1001, 1004 și 1006 nu cunoaștem numărul de telefon, prin urmare, pe cele trei linii, valorile atributului Telefon sunt NULL. Cu toate acestea, interogarea:

SELECT *

FROM clienti

WHERE Telefon IN ('0232212121', NULL)

extrage doar o linie (vezi figura 8.3), pe cea a clientului 1002, deci nici urmă de vreun client cu numărul de telefon necunoscut, deși în listă am inclus și NULL.

CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1002	Client 2 SA	R1002	(null)	700505	0232212121

Figura 8.3. Valorile NULL nu pot fi extrase cu operatorul IN

Iar dacă dorim să extragem clienţii cu adresele nenule, în niciun caz nu folosim operatorul NOT IN, deoarece interogarea:

SELECT *

FROM clienti

WHERE Telefon NOT IN (NULL)

nu extrage nicio linie. După cum vom vedea în paragraful 8.2, în SQL putem folosi în predicatul de selecţie clauza IS NULL, prin care aceste probleme sunt rezolvate destul de uşor.

Tot în paragraful 5.5 am văzut că DB2-ul nu acceptă includerea unui NULL într-o listă-argument a operatorului IN (figura 5.23)[1].

8.1.2. Două tabele suplimentare

Deşi baza de date VÂNZĂRI este alcătuită dintr-un număr considerabil de tabele şi atribute, introducem încă două tabele cu scopul de a gestiona o parte din datele privind drepturile băneşti (salariu negociat şi sporuri) ale angajaţilor firmei. Ambele tabele nu au nimic în comun cu vânzările, ci doar cu salarizarea. Prima tabelă se numeşte PERSONAL2 şi conţine date generale despre angajaţi: marcă; nume şi prenume; data naşterii; compartiment; marca şefului (direct); salariu tarifar. A doua, SPORURI, evidenţiază sporurile lunare primite de fiecare angajat: sporul de vechime (SporVechime), sporul pentru schimbul de noapte (SporNoapte), sporuri pentru condiţii deosebite (SporCD) şi sporuri diverse (AlteSpor).

Cu ajutorul valorii NULL se poate face diferenţa între angajaţii pentru care nu s-a calculat valoarea sporului pe luna curentă (şi care vor avea în câmpul corespunzător valoarea NULL) şi cei care nu au dreptul la un asemenea spor, adică valoarea este 0. Scriptul de creare a celor două tabele este prezentat în listingul 8.1.

Listing 8.1. Script DB2/PostgreSQL de creare a tabelelor SPORURI şi PERSONAL2[2]

```
DROP TABLE sporuri;
DROP TABLE personal2;

CREATE TABLE personal2(
    Marca NUMERIC(5) NOT NULL
        CONSTRAINT pk_personal2 PRIMARY KEY,
    NumePren VARCHAR(40),
    DataNast DATE,
    Compart VARCHAR(20),
    MarcaSef NUMERIC(5)
        CONSTRAINT fk_personal2 REFERENCES personal2(marca),
    SalTarifar NUMERIC(12,2));

CREATE TABLE sporuri(
    An NUMERIC(4) NOT NULL,
    Luna NUMERIC(2) NOT NULL,
```

1. Mesajul de eroare DB2 este, în cazul lansării în execuţie a ultimelor două interogări, *SQL0206N "NULL" is not valid in the context where it is used. SQLSTATE = 42703.*
2. În Oracle, în loc de VARCHAR se foloseşte VARCHAR2, iar în SQL Server, în loc de DATE se foloseşte SMALLDATETIME.

```
Marca NUMERIC(5) NOT NULL
    CONSTRAINT fk_sporuri_pers2 REFERENCES personal2 (marca),
SporVechime NUMERIC(12,2),
SporNoapte NUMERIC(12,2),
SporCD NUMERIC(12,2),
AlteSpor NUMERIC(12,2),
    CONSTRAINT pk_sporuri PRIMARY KEY (an,luna,marca));
```

Figurile 8.4 şi 8.5 ilustrează câteva înregistrări de test ale celor două tabele. Tabela PERSONAL2 conţine zece linii, deci, la momentul curent, firma are zece angajaţi. Pentru doi (angajaţii 4 şi 7) nu li se cunoaşte data naşterii. Cinci angajaţi (2, 4, 5, 6 şi 7) lucrează în compartimentul Financiar, trei (3, 8 şi 9) la compartimentul Marketing, iar unul (10) la Resurse umane. Există şi un angajat care lucrează în cadrul compartimentului Direcţiune. Bănuim că este vorba despre directorul general, bănuială confirmată de faptul că, în linia corespunzătoare acestui angajat (prima), atributul MarcaŞef, care indică marca şefului direct al fiecărui angajat, are valoarea NULL. Cu alte cuvinte, Angajat 1 nu are şef.

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR
1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450
3	ANGAJAT 3	02-08-1962	MARKETING	1	1450
4	ANGAJAT 4	(null)	FINANCIAR	2	1380
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350
7	ANGAJAT 7	(null)	FINANCIAR	5	1280
8	ANGAJAT 8	31-12-1960	MARKETING	3	1290
9	ANGAJAT 9	28-02-1976	MARKETING	3	1410
10	ANGAJAT 10	29-01-1972	RESURSE UMANE	1	1370

Figura 8.4. Câteva înregistrări în tabela PERSONAL2

Dacă privim cu atenţie şi figura 8.5, mai putem bănui că firma s-a înfiinţat în aprilie 2007, când avea numai trei angajaţi. Lucrul acesta ne este sugerat de existenţa, pentru această lună, a numai trei angajaţi pentru care au fost introduse sporuri. La momentul curent (primul trimestru al anului 2008) sunt 10 angajaţi.

AN	LUNA	MARCA	SPORVECHIME	SPORNOAPTE	SPORCD	ALTESPOR
2007	4	1	160	0	0	132
2007	4	2	130	45	0	70
2007	4	3	145	156	420	157
2007	5	1	160	0	0	0
2007	5	2	80	45	0	70
2007	5	3	145	0	0	0
2007	5	10	137	0	0	430
2007	6	1	160	0	0	0
2007	6	2	80	0	0	150
2007	6	4	50	15	88	120
2007	6	5	130	15	0	20
2007	6	10	200	12	0	6
2007	7	1	160	0	(null)	(null)
2007	7	2	80	0	0	158
2007	7	3	145	0	0	0
2007	7	4	50	15	(null)	15
2007	7	5	130	0	0	120
2007	7	6	110	147	0	0
2007	7	7	60	210	0	0
2007	7	8	130	0	15	0
2007	7	9	140	100	77	0
2007	7	10	200	0	0	120

Figura 8.5. Câteva înregistrări în tabela SPORURI

Popularea celor trei tabele este conţinută în listingul 8.2.

Listing 8.2. Script Oracle/PostgreSQL de populare a tabelelor SPORURI şi PERSONAL2[3]

```
DELETE FROM personal2;
INSERT INTO personal2 VALUES (1, 'ANGAJAT 1', DATE'1962-07-01',
    'DIRECTIUNE', NULL, 1600);
INSERT INTO personal2 VALUES (2, 'ANGAJAT 2', DATE'1977-10-11',
    'FINANCIAR', 1, 1450);
INSERT INTO personal2 VALUES (3, 'ANGAJAT 3', DATE'1962-08-02',
    'MARKETING', 1, 1450);
INSERT INTO personal2 VALUES (4, 'ANGAJAT 4', NULL,
    'FINANCIAR', 2, 1380);
INSERT INTO personal2 VALUES (5, 'ANGAJAT 5', DATE'1965-04-30',
    'FINANCIAR', 2, 1420);
INSERT INTO personal2 VALUES (6, 'ANGAJAT 6', DATE'1965-11-09',
    'FINANCIAR', 5, 1350);
INSERT INTO personal2 VALUES (7, 'ANGAJAT 7', NULL, 'FINANCIAR', 5, 1280);
INSERT INTO personal2 VALUES (8, 'ANGAJAT 8', DATE'1960-12-31',
    'MARKETING', 3, 1290);
INSERT INTO personal2 VALUES (9, 'ANGAJAT 9', DATE'1976-02-28',
    'MARKETING', 3, 1410);
INSERT INTO personal2 VALUES (10, 'ANGAJAT 10', DATE'1972-01-29',
    'RESURSE UMANE', 1, 1370);

DELETE FROM sporuri ;
INSERT INTO sporuri VALUES (2007, 4, 1, 160, 0, 0, 132);
```

3. Versiunea DB2/SQL Server a scriptului se obţine prin eliminarea cuvintelor DATE.

```
INSERT INTO sporuri VALUES (2007, 4, 2, 130, 45, 0, 70);
INSERT INTO sporuri VALUES (2007, 4, 3, 145, 156, 420, 157);
INSERT INTO sporuri VALUES (2007, 5, 1, 160, 0, 0, 0);
INSERT INTO sporuri VALUES (2007, 5, 2, 80, 45, 0, 70);
INSERT INTO sporuri VALUES (2007, 5, 3, 145, 0, 0, 0);
INSERT INTO sporuri VALUES (2007, 5, 10, 137, 0, 0, 430);
INSERT INTO sporuri VALUES (2007, 6, 1, 160, 0, 0, 0);
INSERT INTO sporuri VALUES (2007, 6, 2, 80, 0, 0, 150);
INSERT INTO sporuri VALUES (2007, 6, 4, 50, 15, 88, 120);
INSERT INTO sporuri VALUES (2007, 6, 5, 130, 15, 0, 20);
INSERT INTO sporuri VALUES (2007, 6, 10, 200, 12, 0, 6);
INSERT INTO sporuri VALUES (2007, 7, 1, 160, 0, NULL, NULL);
INSERT INTO sporuri VALUES (2007, 7, 2, 80, 0, 0, 158);
INSERT INTO sporuri VALUES (2007, 7, 3, 145, 0, 0, 0);
INSERT INTO sporuri VALUES (2007, 7, 4, 50, 15, NULL, 15);
INSERT INTO sporuri VALUES (2007, 7, 5, 130, 0, 0, 120);
INSERT INTO sporuri VALUES (2007, 7, 6, 110, 147, 0, 0);
INSERT INTO sporuri VALUES (2007, 7, 7, 60, 210, 0, 0);
INSERT INTO sporuri VALUES (2007, 7, 8, 130, 0, 15, 0);
INSERT INTO sporuri VALUES (2007, 7, 9, 140, 100, 77, 0);
INSERT INTO sporuri VALUES (2007, 7, 10, 200, 0, 0, 120);
```

8.2. Operatorul IS NULL

Valoarea NULL nu se confundă cu valoarea 0 - pentru atributele numerice - sau cu valoarea ' ' - spaţiu - pentru atributele de tip şir de caractere. Este important de notat că, în vederea identificării valorilor nule, operatorul are forma *IS NULL* şi nu *= NULL*.

Pentru care dintre clienţi nu se cunoaşte adresa?

Soluţia se bazează pe utilizarea operatorului IS NULL care extrage toate valorile NULL pentru un atribut (figura 8.6):

```
SELECT *
FROM clienti
WHERE Adresa IS NULL
```

CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1002	Client 2 SA	R1002	(null)	700505	0232212121
1005	Client 5 SRL	R1005	(null)	701900	0256111111

Figura 8.6. Clienţii „fără adresă"

Prin execuţia frazei SQL:

```
SELECT *
FROM clienti
WHERE Adresa = NULL
```

se va obţine o tabelă cu 0 (zero) linii sau chiar o eroare (în DB2[4]). Rezultatul evaluării **Adresa = NULL** va fi întotdeauna FALSE.

În unele SGBD-uri, precum Oracle, există o sursă de confuzie între şirul de caractere de lungime zero ('', adică două apostrofuri lipite, fără spaţiu) şi NULL-ul propriu-zis. Astfel, dacă vrem să NULL-izăm adresa clientului cu codul 1007 putem recurge la varianta următoare a comenzii UPDATE:

```
UPDATE clienti
SET Adresa = ''
WHERE CodCl=1007
```

Atribuirea s-a făcut, deci, cu **Adresa = ''**, şi nu cu **Adresa = NULL**. Rezultatul este acelaşi, dacă examinăm linia acestui client după comanda UPDATE (vezi figura 8.7).

CODCL	DENCL	CODFISCAL	ADRESA	CODPOST	TELEFON
1007	Client 7 SRL	R1007	(null)	701900	0256121212

Figura 8.7. NULL-itatea adresei clientului 1007 (în Oracle)

Prin urmare, dacă şirul vid '' (de lungime zero) este echivalentul valorii NULL, la atribuire, nu la fel stau lucrurile în clauza WHERE, întrucât interogarea Oracle:

```
SELECT *
FROM CLIENTI
WHERE Adresa = ''
```

„continuă” să nu extragă nicio linie. Lucrurile nu stau la fel în DB2 şi în MS SQL Server, întrucât interogarea extrage linia corespunzătoarea clientului 1007, valoarea adresei pentru acest client nefiind considerată nulă. În concluzie, dacă pentru DB2, PostgreSQL şi SQL Server, şirul de caractere de lungime zero – '' (două apostrofuri, nu ghilimele!) – este diferit de NULL, pentru Oracle uneori este diferit (în clauza WHERE), alteori acelaşi lucru (în clauza SET a comenzii UPDATE).

Care sunt persoanele şi lunile pentru care nu s-a calculat (nu se cunoaşte) *sporul pentru condiţii deosebite?*

Prin interogarea:

```
SELECT sporuri.Marca, NumePren, Compart, An, Luna
FROM personal2
INNER JOIN sporuri ON personal2.Marca = sporuri.Marca
WHERE SporCD IS NULL
```

se obţine situaţia din figura 8.8.

MARCA	NUMEPREN	COMP...	AN	LUNA
1	ANGAJAT 1	DIRECTIUNE	2007	7
4	ANGAJAT 4	FINANCIAR	2007	7

Figura 8.8. Angajaţii pentru care nu s-au operat sporurile pentru condiţii deosebite

4. Mesajul de eroare din DB2 la execuţia comenzii este: *SQL0401N The data types of the operands for the operation "=" are not compatible. SQLSTATE=42818.*

Care sunt angajații și lunile în care aceștia nu au primit *spor pentru condiții deosebite?*

Atât soluția, cât și rezultatul sunt sensibil diferite (vezi figura 8.9), întrucât acum ne interesează valorile zero (nu NULL) ale SporCD:

```
SELECT sporuri.Marca, NumePren, Compart, An, Luna
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE SporCD = 0
ORDER BY An, Luna, NumePren
```

MARCA	NUMEPREN	COMPART	AN	LUNA
1	ANGAJAT 1	DIRECTIUNE	2007	4
2	ANGAJAT 2	FINANCIAR	2007	4
1	ANGAJAT 1	DIRECTIUNE	2007	5
10	ANGAJAT 10	RESURSE UMANE	2007	5
2	ANGAJAT 2	FINANCIAR	2007	5
3	ANGAJAT 3	MARKETING	2007	5
1	ANGAJAT 1	DIRECTIUNE	2007	6
10	ANGAJAT 10	RESURSE UMANE	2007	6
2	ANGAJAT 2	FINANCIAR	2007	6
5	ANGAJAT 5	FINANCIAR	2007	6
10	ANGAJAT 10	RESURSE UMANE	2007	7
2	ANGAJAT 2	FINANCIAR	2007	7
3	ANGAJAT 3	MARKETING	2007	7
5	ANGAJAT 5	FINANCIAR	2007	7
6	ANGAJAT 6	FINANCIAR	2007	7
7	ANGAJAT 7	FINANCIAR	2007	7

Figura 8.9. Angajații și lunile pentru care SporCD este zero (neNULL)

Care dintre angajați sunt născuți înainte de 1 ianuarie 1970 și care după această dată?

Persoanele născute *înainte* (figura 8.10) se obțin prin:

```
SELECT *
FROM personal2
WHERE DataNast < DATE'1970-01-01'
```

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR
1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600
3	ANGAJAT 3	02-08-1962	MARKETING	1	1450
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350
8	ANGAJAT 8	31-12-1960	MARKETING	3	1290

Figura 8.10. Persoane născute înainte de 1 ianuarie 1970

iar cele născute *după* 1 ianuarie 1970 (figura 8.11) prin:

```
SELECT *
FROM personal2
WHERE DataNast >= DATE'1970-01-01'
```

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450
9	ANGAJAT 9	28-02-1976	MARKETING	3	1410
10	ANGAJAT 10	29-01-1972	RESURSE UMANE	1	1370

Figura 8.11. Persoane născute după 1 Ianuarie 1970

Dacă reunim mulţimea angajaţilor născuţi înainte de data fixată cu mulţimea celor născuţi după această dată nu obţinem relaţia iniţială PERSONAL2 - vezi figura 8.12!

```
SELECT * FROM personal2
WHERE DataNast < DATE'1970-01-01'
    UNION
SELECT * FROM personal2
WHERE DataNast >= DATE'1970-01-01'
```

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR
1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450
3	ANGAJAT 3	02-08-1962	MARKETING	1	1450
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350
8	ANGAJAT 8	31-12-1960	MARKETING	3	1290
9	ANGAJAT 9	28-02-1976	MARKETING	3	1410
10	ANGAJAT 10	29-01-1972	RESURSE UMANE	1	1370

Figura 8.12. Reuniunea persoanelor născute înainte de 1 ianuarie 1970 cu persoanele născute după această dată

Din tabela obţinută în figura 8.12 lipsesc angajaţii care nu au precizată data naşterii, altfel spus, persoanele „fără vârstă". Pentru recompunerea tabelei PERSONAL2, în reuniune mai trebuie adăugate şi liniile pentru care *DataNast IS NULL*, adică:

```
SELECT * FROM personal2 WHERE DataNast < DATE'1970-01-01'
    UNION
SELECT * FROM personal2 WHERE DataNast >= DATE'1970-01-01'
    UNION
SELECT * FROM personal2 WHERE DataNast IS NULL
ORDER BY 1
```

Revenind la exemplele din finalul paragrafului 5.5, interogarea *SELECT * FROM facturi WHERE DataFact IN (DATE'2007-08-01', DATE'2007-08-03', DATE'2007-08-*

07', NULL) este echivalentă cu *SELECT * FROM facturi WHERE DataFact IN (DATE'2007-08-01', DATE'2007-08-03', DATE'2007-08-07')* în Oracle, PostgreSQL şi SQL Server (nu uitaţi să eliminaţi DATE-urile în SQL Server), deoarece logica execuţiei sale este, de fapt:

```
SELECT *
FROM facturi
WHERE DataFact = DATE'2007-08-01' OR DataFact = DATE'2007-08-03'
      OR DataFact = DATE'2007-08-07' OR DataFact =NULL
```

iar rezultatul unui *sau logic* (OR) dintre o valoare TRUE şi NULL este TRUE (vezi tabelul 8.1).

Tabelul 8.1. Combinarea a două expresii utilizând operatorul logic OR

OR	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

În schimb, interogarea *SELECT * FROM facturi WHERE DataFact NOT IN (DATE'2007-08-01', DATE'2007-08-03', DATE'2007-08-07', NULL)* nu returnează nicio linie, întrucât este echivalentă cu:

```
SELECT *
FROM facturi
WHERE DataFact <> DATE'2007-08-01' AND DataFact <> DATE'2007-08-03'
      AND DataFact <> DATE'2007-08-07' AND DataFact <> NULL
```

iar rezultatul unui *şi logic* (AND) dintre o valoare TRUE şi NULL este NULL (vezi tabelul 8.2). Discuţia nu este aplicabilă în DB2, care nu tolerează NULL-uri în liste (ştim asta din capitolul 5).

Tabelul 8.2. Combinarea a două expresii utilizând operatorul logic AND

AND	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Pentru negaţia logică (NOT), evaluarea se face ca în tabelul 8.3. Probleme similare apar şi la interogări care folosesc subconsultări, ce prezintă valori nule (vezi capitolul 9).

Tabelul 8.3. Rezultatele utilizării operatorului logic NOT

NOT	**TRUE**	**FALSE**	**UNKNOWN**
	FALSE	TRUE	UNKNOWN

Încheiem paragraful abordând comportamentul SQL-ului în situaţiile în care „subiectul predicatului" IS NULL nu este un singur atribut, ca până acum, ci două

sau mai multe atribute. Problema a fost atacată de Melton şi Simon ([Melton & Simon 2002], pp. 188-189). În SQL, interogarea:

```
SELECT *
FROM sporuri
WHERE (SporCD, AlteSpor) IS NULL
```

este echivalentă cu:

```
SELECT *
FROM sporuri
WHERE SporCD IS NULL AND AlteSpor IS NULL
```

rezultatul fiind cel din figura 8.13. Prin urmare, valorile ambelor atribute trebuie să fie, simultan, NULL.

AN	LUNA	MARCA	SPORVECHIME	SPORNOAPTE	SPORCD	ALTESPOR
2007	7	1	160	0	(null)	(null)

Figura 8.13. Predicat IS NULL aplicat asupra a două atribute

Genul de predicat în care argumentul operatorului IS NULL este un tuplu (alcătuit din două sau mai multe atribute) „ad-hoc" nu funcţionează nici în Oracle, nici în MS SQL Server, nici în DB2, însă, surprinzător (pentru un server BD *open source*) merge în PostgreSQL! De aceea, pentru cele trei servere BD ultima interogare reprezintă o soluţie viabilă.

Analog discuţiei de mai sus,

```
SELECT *
FROM sporuri
WHERE (SporCD, AlteSpor) IS NOT NULL
```

este echivalent cu:

```
SELECT *
FROM sporuri
WHERE SporCD IS NOT NULL AND AlteSpor IS NOT NULL
```

(vezi figura 8.14) pentru că încearcă să extragă toate liniile în care valorile ambelor atribute sunt simultam nenule şi, tot la fel de analog, prima variantă nu funcţionează în DB2, Oracle şi SQL Server.

	an numeric(4,0)	luna numeric(2,0)	marca numeric(5,0)	sporvechime numeric(12,2	spornoapte numeric(12,2	sporcd numeric(12,2	altespor numeric(12,2
1	2007	4	1	160.00	0.00	0.00	132.00
2	2007	4	2	130.00	45.00	0.00	70.00
3	2007	4	3	145.00	156.00	420.00	157.00
4	2007	5	1	160.00	0.00	0.00	0.00
5	2007	5	2	80.00	45.00	0.00	70.00
6	2007	5	3	145.00	0.00	0.00	0.00
7	2007	5	10	137.00	0.00	0.00	430.00
8	2007	6	1	160.00	0.00	0.00	0.00
9	2007	6	2	80.00	0.00	0.00	150.00
10	2007	6	4	50.00	15.00	88.00	120.00
11	2007	6	5	130.00	15.00	0.00	20.00
12	2007	6	10	200.00	12.00	0.00	6.00
13	2007	7	2	80.00	0.00	0.00	158.00
14	2007	7	3	145.00	0.00	0.00	0.00
15	2007	7	5	130.00	0.00	0.00	120.00
16	2007	7	6	110.00	147.00	0.00	0.00
17	2007	7	7	60.00	210.00	0.00	0.00
18	2007	7	8	130.00	0.00	15.00	0.00
19	2007	7	9	140.00	100.00	77.00	0.00
20	2007	7	10	200.00	0.00	0.00	120.00

Figura 8.14. Predicatul *(SporCD, AlteSpor) IS NOT NULL*

Lucrurile se complică un pic în PostgreSQL (în celelalte trei SGBD-uri luăm, momentan, o pauză), dacă negația este plasată înaintea tuplului „ad-hoc". Să începem cu interogarea:

```
SELECT *
FROM sporuri
WHERE NOT (SporCD, AlteSpor) IS NULL
```

Rezultatul din figura 8.15 conține 21 de linii. Față de figura 8.14, linia în plus (a 15-a) este cea în care valoarea atributului SporCD este nulă, iar cea a atributului AlteSpor nenulă. Motivul este că interogarea de mai sus este echivalentă cu:

```
SELECT *
FROM sporuri
WHERE (NOT SporCD IS NULL) OR (NOT AlteSpor IS NULL)
```

care, mai departe, înseamnă:

```
SELECT *
FROM sporuri
WHERE (SporCD IS NOT NULL) OR (AlteSpor IS NOT NULL)
```

(parantezele sunt pur explicative).

	an numeric(4,0)	luna numeric(2,0)	marca numeric(5,0)	sporvechime numeric(12,2	spornoapte numeric(12,2	sporcd numeric(12,2	altespor numeric(12,2
1	2007	4	1	160.00	0.00	0.00	132.00
2	2007	4	2	130.00	45.00	0.00	70.00
3	2007	4	3	145.00	156.00	420.00	157.00
4	2007	5	1	160.00	0.00	0.00	0.00
5	2007	5	2	80.00	45.00	0.00	70.00
6	2007	5	3	145.00	0.00	0.00	0.00
7	2007	5	10	137.00	0.00	0.00	430.00
8	2007	6	1	160.00	0.00	0.00	0.00
9	2007	6	2	80.00	0.00	0.00	150.00
10	2007	6	4	50.00	15.00	88.00	120.00
11	2007	6	5	130.00	15.00	0.00	20.00
12	2007	6	10	200.00	12.00	0.00	6.00
13	2007	7	2	80.00	0.00	0.00	158.00
14	2007	7	3	145.00	0.00	0.00	0.00
15	2007	7	4	50.00	15.00		15.00
16	2007	7	5	130.00	0.00	0.00	120.00
17	2007	7	6	110.00	147.00	0.00	0.00
18	2007	7	7	60.00	210.00	0.00	0.00
19	2007	7	8	130.00	0.00	15.00	0.00
20	2007	7	9	140.00	100.00	77.00	0.00
21	2007	7	10	200.00	0.00	0.00	120.00

Figura 8.15. Predicatul *NOT (SporCD, AlteSpor) IS NULL*

Pentru ca ameţeala să fie completă, mai trebuie să adaugăm că:

```
SELECT *
FROM sporuri
WHERE NOT (SporCD, AlteSpor) IS NOT NULL
```

(figura 8.16) este echivalentă:

```
SELECT *
FROM sporuri
WHERE (NOT SporCD IS NOT NULL) OR
        (NOT AlteSpor IS NOT NULL)
```

care, la rândul său, este echivalentă cu mult mai simpla versiune:

```
SELECT * FROM sporuri
WHERE (SporCD IS NULL) OR (AlteSpor IS NULL)
```

	an numeric(4,0)	luna numeric(2,0)	marca numeric(5,0)	sporvechime numeric(12,2	spornoapte numeric(12,2	sporcd numeric(12,2	altespor numeric(12,2
1	2007	7	1	160.00	0.00		
2	2007	7	4	50.00	15.00		15.00

Figura 8.16. Predicatul *NOT (SporCD, AlteSpor) IS NOT NULL*

Negaţia la nivel de tuplu este, de cele mai multe ori, un simplu exerciţiu de preţiozitate, mai ales când discutăm despre negarea negaţiei (nicio legătură cu Engels!, deşi recenta criză mondială a adus o nesperată culoare în obrajii [neo-] marxiştilor).

8.3. Transformarea NULL-ilor

Am văzut că un prim aspect iritant al NULL-ilor în SQL se manifestă la includerea acestei valori într-o listă precedată de operatorul NOT IN. Probabil, însă că iritarea o să crească dacă luăm câteva exemple ce conţin expresii în care se strecoară câte un NULL.

Care este totalul sporurilor fiecărui angajat pe luna iulie 2007?

```
SELECT sporuri.Marca, NumePren, Compart,
        SporVechime + SporNoapte + SporCD + AlteSpor
            AS TotalSporuri
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE An = 2007 AND Luna = 7
```

Din păcate, rezultatul este incorect (vezi figura 8.17), deoarece, din prezentarea conţinutului tabelei SPORURI (figura 8.5), reiese că, pe luna iulie 2007, ANGAJAT 1 are sporul de vechime calculat, iar ANGAJAT 4 are, pe aceeaşi lună, şi spor de vechime, şi de noapte, şi alte sporuri, acestea nefiind luate în calcul la însumare.

MARCA	NUMEPREN	COMPART	TOTALSPORURI
1	ANGAJAT 1	DIRECTIUNE	(null)
2	ANGAJAT 2	FINANCIAR	238
3	ANGAJAT 3	MARKETING	145
4	ANGAJAT 4	FINANCIAR	(null)
5	ANGAJAT 5	FINANCIAR	250
6	ANGAJAT 6	FINANCIAR	257
7	ANGAJAT 7	FINANCIAR	270
8	ANGAJAT 8	MARKETING	145
9	ANGAJAT 9	MARKETING	317
10	ANGAJAT 10	RESURSE UMANE	320

Figura 8.17. Rezultatul unei expresii în care cel puţin un operand este NULL

Explicaţia este simplă: dacă într-o expresie unul dintre operanzi este NULL, atunci rezultatul evaluării întregii expresii este NULL. Fac excepţie funcţiile -agregat (paragraful 6.7). Dacă, spre exemplu, vrem să calculăm *totalul sporurilor de noapte acordate pentru luna iulie 2007*, fraza:

```
SELECT SUM(SporNoapte) AS Total_SporuriNoapte_Luna_7
FROM sporuri
WHERE An = 2007 AND Luna = 7
```

calculează corect rezultatul (figura 8.18).

TOTAL_SPORURINOAPTE_LUNA_7
472

Figura 8.18. Funcţia SUM (SporNoapte) nu este afectată de eventualele valori NULL

Revenim la problema din figura 8.17. Pentru a asigura corectitudinea totalului, ar trebui ca, în expresie, orice valoare nulă să fie considerată zero. Lucru realizabil, deoarece încă din SQL92 a apărut o funcţie în acest sens - COALESCE:

```
SELECT sporuri.Marca, NumePren, Compart,
        COALESCE(SporVechime,0) +
        COALESCE(SporNoapte,0) + COALESCE(SporCD,0) +
        COALESCE(AlteSpor,0) AS TotalSporuri
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE An = 2007 AND Luna = 7
```

Sumele obţinute sunt în acest caz cele din figura 8.19.

MARCA	NUMEPREN	COMPART	TOTALSPORURI
1	ANGAJAT 1	DIRECTIUNE	160
2	ANGAJAT 2	FINANCIAR	238
3	ANGAJAT 3	MARKETING	145
4	ANGAJAT 4	FINANCIAR	80
5	ANGAJAT 5	FINANCIAR	250
6	ANGAJAT 6	FINANCIAR	257
7	ANGAJAT 7	FINANCIAR	270
8	ANGAJAT 8	MARKETING	145
9	ANGAJAT 9	MARKETING	317
10	ANGAJAT 10	RESURSE UMANE	320

Figura 8.19. Conversia valorilor nule în zero şi evaluarea corectă a expresiei

Este important de reţinut că funcţia nu se aplică la nivel de expresie, ci fiecărui operand susceptibil de nulitate. Completăm SELECT-ul anterior cu o coloană în care funcţia se aplică la nivelul întregii expresii:

```
SELECT sporuri.Marca, NumePren, Compart,
        COALESCE(SporVechime,0) + COALESCE (SporNoapte,0) +
        COALESCE(SporCD,0) + COALESCE (AlteSpor,0)
                AS TotalSporuri_Corect,
        COALESCE(SporVechime + SporNoapte + SporCD +
                AlteSpor,0) AS TotalSporuri_Incorect
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE An = 2007 AND Luna = 7
```

Rezultatul din figura 8.20 arată destul de limpede diferenţa pentru angajaţii 1 şi 4, ei fiind cei care prezintă sporuri cu valori NULL pentru iulie 2007.

Am aplicat funcţia COALESCE după nevoile exemplului. Formatul său general este însă ceva mai generos. Să presupunem că pentru luna iulie 2007, în urma unor discuţii aprinse, s-a hotărât ca, la persoanele cărora nu li s-a calculat sporul pentru

condiţii deosebite (SporCD), valoarea acestuia să fie egală cu alte sporuri (AlteSpor); dacă nici alte sporuri nu s-au calculat, atunci valoarea luată în considerare să fie zero.

MARCA	NUMEPREN	COMPART	TOTALSPORURI_CORECT	TOTALSPORURI_INCORECT
1	ANGAJAT 1	DIRECTIUNE	160	0
2	ANGAJAT 2	FINANCIAR	238	238
3	ANGAJAT 3	MARKETING	145	145
4	ANGAJAT 4	FINANCIAR	80	0
5	ANGAJAT 5	FINANCIAR	250	250
6	ANGAJAT 6	FINANCIAR	257	257
7	ANGAJAT 7	FINANCIAR	270	270
8	ANGAJAT 8	MARKETING	145	145
9	ANGAJAT 9	MARKETING	317	317
10	ANGAJAT 10	RESURSE UMANE	320	320

Figura 8.20. Funcția COALESCE aplicată la nivel de operand şi la nivelul întregii expresii

Figura 8.21 clarifică puţin lucrurile. Cu excepţia rândurilor corespunzătoare angajaţilor cu mărcile 1 şi 4, valorile din coloana a şasea (COALESCE...) sunt egale cu valori SporCD. Pentru angajatul cu marca 4, întrucât SporCD este nul, valoarea de pe coloana a şasea va prelua valoarea atributului AlteSpor (15). La angajatul cu marca 1, SporCd şi AlteSpor sunt nule, aşa încât pe a şasea coloană apare valoarea zero.

MARCA	SpVech	SpNoapte	SPORCD	ALTESPOR	COALESCE(SPORCD,ALTESPOR,0)	Total_Nou
1	160	0	(null)	(null)	0	160
2	80	0	0	158	0	238
3	145	0	0	0	0	145
4	50	15	(null)	15	15	95
5	130	0	0	120	0	250
6	110	147	0	0	0	257
7	60	210	0	0	0	270
8	130	0	15	0	15	145
9	140	100	77	0	77	317
10	200	0	0	120	0	320

Figura 8.21. Când valoarea SporCD e NULL, se preia valoarea AlteSpor; când şi aceasta e NULL, se consideră zero

Acest rezultat se obţine tot cu funcţia COALESCE, dar cu sintaxa următoare:

```
SELECT Marca,
        SporVechime AS "SpVech",
        SporNoapte AS "SpNoapte",
        SporCD,
        AlteSpor,
        COALESCE (SporCD, AlteSpor, 0),
```

```
        COALESCE(SporVechime,0) + COALESCE(SporNoapte,0) +
        COALESCE(SporCD, AlteSpor, 0) +
        COALESCE(AlteSpor,0)
                        AS "Total_Nou"
FROM sporuri
WHERE An = 2007 AND Luna = 7
```

Prin urmare, COALESCE poate avea două sau mai multe argumente, returnând prima valoare nenulă întâlnită (parcurgerea are loc în ordinea enumerării). Toate interogările din acest paragraf sunt corecte în toate cele patru dialecte.

După cum valoarea NULL este una universal aplicabilă (pot avea valori NULL şi atribute numerice, şi atribute de şir, dată calendaristică etc.), tot aşa funcţia COALESCE poate fi aplicată asupra oricărui de argument. Revenim la un exemplu din paragraful 5.4.

Să se afişeze facturile din luna septembrie 2007 în ordinea alfabetică a observaţiilor (valorile atributul *Obs*).

În Oracle rezolvam problema ordonării cu ajutorul opţiunilor NULLS FIRST sau NULLS LAST. Spuneam atunci că PostgreSQL-ul (până la versiunea 8.3), MS SQL Server şi DB2 fac nazuri, neavând implementate aceste opţiuni. În acest moment suntem în măsură să formulăm soluţii bazate pe COALESCE care transformă valorile NULL în spaţii, spaţii ce apar la ordonare înaintea literelor (mari sau mici):

```
SELECT *
FROM facturi
WHERE datafact >= DATE'2007-09-01' AND datafact <= DATE'2007-09-30'
ORDER BY COALESCE(obs,'') – NULLS FIRST
```

Aceeaşi interogare, din care scoatem cele două cuvinte DATE, rezolvă problema şi în DB2, şi în MS SQL Server. Dacă am fi intenţionat ca liniile cu valori NULL ale atributului Obs să apară la finalul rezultatului (NULLS LAST) conversia trebuia făcută într-un şir de caractere (zzz) care apare, garantat, la sfârşit:

```
SELECT * FROM facturi
WHERE datafact >= DATE'2007-09-01' AND datafact <= DATE'2007-09-30'
ORDER BY COALESCE (obs, 'zzz') – NULLS LAST
```

Ne interesează, în continuare, să analizăm observaţiile legate de facturi, în sensul numărării de câte ori apare o anumită observaţie. Tabela care ne interesează este FACTURI, iar gruparea se realizează după valorile atributului Obs:

```
SELECT Obs, COUNT(*)
FROM facturi
GROUP BY Obs
ORDER BY Obs NULLS LAST
```

Rezultatul (Oracle) se prezintă ca în partea stângă a figurii 8.22. Pentru grupul facturilor fără observaţii, vrem ca în loc de NULL să apară, tot la final, indicaţia * *fără observaţii** (vezi partea dreaptă a figurii 8.22); se foloseşte funcţia COALESCE astfel:

OBS	COUNT(*)
Pretul propus initial a fost modificat	3
Probleme cu transportul	3
(null)	24

COALESCE(OBS,'*FARAOBSERVATII*')	COUNT(*)
Pretul propus initial a fost modificat	3
Probleme cu transportul	3
* fara observatii *	24

Figura 8.22. Folosirea COALESCE pentru un argument şir de caractere

```
SELECT COALESCE(Obs,'* fara observatii *'), COUNT(*)
FROM facturi
GROUP BY Obs
ORDER BY Obs NULLS LAST
```

sau, pentru a funcţiona şi PostgreSQL pre8.3, DB2 şi SQL Server:

```
SELECT COALESCE(Obs,'* fara observatii *'), COUNT(*)
FROM facturi
GROUP BY Obs
ORDER BY COALESCE(Obs,'zzz fara observatii *')
```

Dacă am dori să extragem din tabela PERSONAL2 persoanele născute înainte de 1 ianuarie 1970 şi cele pentru care nu cunoaştem data naşterii, ordonate după data naşterii, până în acest paragraf puteam formula doar o soluţie bazată pe UNION:

```
SELECT *
FROM personal2
WHERE DataNast < DATE'1970-01-01'
        UNION
SELECT *
FROM personal2
WHERE DataNast IS NULL
ORDER BY 3 NULLS FIRST
```

Acum însă putem renunţa la UNION beneficiind de COALESCE, interogarea funcţionând şi în PostgreSQL (şi SQL Server, dacă eliminăm DATE):

```
SELECT *
FROM personal2
WHERE COALESCE (DataNast,DATE'1950-01-01' ) < DATE'1970-01-01'
ORDER BY COALESCE (DataNast, DATE'1950-01-01')
```

În unele dialecte SQL, funcţionalitatea funcţiei COALESCE este asigurată de VALUE (de exemplu, DB2) şi NVL (Oracle, Visual FoxPro şi, începând cu versiunea 9.5, DB2). Funcţia NVL din Oracle are numai două argumente, valoarea returnată fiind a primului argument, dacă acesta este nenul, sau a celui de-al doilea (când valoarea primului este NULL). Atunci când numărul argumentelor este trei, se poate folosi o funcţie soră (vitregă) - NVL2 care funcţionează pe acelaşi principiu. NVL şi COALESCE sunt sinonime în DB2, atât ca funcţionalitate, cât şi ca sintaxă.

Un alt element interesant legat de valorile nule ţine de conversia în sens invers, dintr-o valoare oarecare în NULL.

Să se determine totalul sporurilor de vechime pentru luna iulie 2007, dar în rezultat să nu fie luate în calcul valorile (valoarea, dacă apare o singură dată) 130 RON.

Soluţia „clasică" este:

```
SELECT SUM(SporVechime)
FROM sporuri
WHERE An = 2007 AND Luna = 7 AND SporVechime <> 130
```

O variantă ceva mai elegantă se redactează prin utilizarea funcţiei NULLIF prezentă încă din SQL-92 care, în interogarea de mai jos, converteşte orice apariţie a valorii în NULL[5]:

```
SELECT SUM(NULLIF(SporVechime,130))
FROM sporuri
WHERE An = 2007 AND Luna = 7
```

Pentru acest gen de probleme se pot folosi şi structuri de tip CASE (paragraful 8.5) sau funcţii „proprietare", cum ar fi REPLACE sau DECODE în Oracle[6]:

```
SELECT SUM(REPLACE(SporVechime,130,NULL))
FROM sporuri
WHERE  An = 2007 AND Luna = 7
```

8.4. Joncţiunea externă

Standardul SQL-92 a introdus operatorii necesari joncţiunii externe pe care am prezentat-o în paragraful 4.4.5 (figura 4.27):

- LEFT OUTER JOIN pentru joncţiune externă la stânga;
- RIGHT OUTER JOIN pentru joncţiune externă la dreapta;
- FULL OUTER JOIN pentru joncţiune externă totală (în ambele direcţii)[7].

Dacă ne raportăm la exemplul teoretic din algebra relaţională, atunci joncţiunile externe la stânga, la dreapta şi totală dintre relaţiile R1 şi R2 se transcriu în SQL astfel:

• joncţiune externă la stânga:

```
SELECT *
FROM r1 LEFT OUTER JOIN r2 ON r1.C = r2.C
```

• joncţiune externă la dreapta:

---

5. Ambele soluţii returnează, pentru conţinutul curent al tabelelor, valoarea 945 (prin comparaţie cu 1205 care este totalul sporurilor de vechime pe luna iulie 2007).
6. Funcţia DECODE a fost introdusă şi în DB2, versiunea 9.5, însă nu funcţionează în acest exemplu, din cauza existenţei valorii NULL în lista-argument.
7. Toate cele patru servere BD acceptă omiterea cuvântului OUTER pentru orice tip de joncţiune externă: LEFT JOIN, RIGHT JOIN, FULL JOIN. Însă noi vom folosi titulatura completă, pentru claritate.

SELECT *

FROM r1 RIGHT OUTER JOIN r2 ON r1.C = r2.C

- joncţiune externă totală:

SELECT *

FROM r1 FULL OUTER JOIN r2 ON r1.C = r2.C

Următorul exemplu este desprins tot din algebra relaţională (paragraful 4.4.5, exemplul 21).

Care sunt codurile poştale care nu fac parte din adresa niciunui client?

Prin joncţiunea externă la stânga dintre CODURI_POŞTALE şi CLIENŢI:

```
SELECT *
FROM coduri_postale LEFT OUTER JOIN clienti
        ON coduri_postale.CodPost = clienti.CodPost
ORDER BY 1
```

se obţine tabela din figura 8.23. Primele trei coloane corespund tabelei CODURI_POŞTALE, iar următoarele (ultimele) şase, tabelei CLIENŢI. Liniile puse în evidenţă ţin strict de joncţionarea externă la stânga, deoarece prezintă valori nenule numai pentru primele trei atribute (din CODURI_POŞTALE), completate cu valori NULL pentru toate atributele preluate din CLIENŢI. Joncţiunea internă (echi- sau naturală) ar fi extras numai liniile nemarcate.

CODPOST	LOC	JUD	CODCL	DENCL	CODFISCAL	ADRESA	CODPOST_1	TELEFON
700505	Iasi	IS	1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	(null)
700505	Iasi	IS	1002	Client 2 SA	R1002	(null)	700505	0232212121
700510	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
700515	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
701150	Pascani	IS	1004	Client 4	(null)	Sapientei, 56	701150	(null)
701900	Timisoara	TM	1007	Client 7 SRL	R1007	Victoria Capitalismului, 2	701900	0256121212
701900	Timisoara	TM	1005	Client 5 SRL	R1005	(null)	701900	0256111111
705300	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705310	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705550	Roman	NT	1006	Client 6 SA	R1006	Pacientei, 33	705550	(null)
705800	Suceava	SV	(null)	(null)	(null)	(null)	(null)	(null)
706400	Birlad	VS	(null)	(null)	(null)	(null)	(null)	(null)
706500	Vaslui	VS	1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	0235222222
706510	Vaslui	VS	(null)	(null)	(null)	(null)	(null)	(null)

Figura 8.23. Joncţiunea externă la stânga dintre CODURI_POŞTALE şi CLIENŢI

Pentru că în rezultat ne interesează numai codurile poştale care nu se regăsesc în adresa niciunui client, se va folosi o clauză WHERE în care unul dintre atributele (de preferinţă cheia primară sau, oricum, un atribut NOT NULL) preluate din CLIENŢI este NULL:

SELECT *

FROM coduri_postale

```
	LEFT OUTER JOIN clienti ON coduri_postale.CodPost = clienti.CodPost
WHERE clienti.CodPost IS NULL
ORDER BY 1
```

CODPOST	LOC	JUD	CODCL	DENCL	CODFISCAL	ADRESA	CODPOST_1	TELEFON
700510	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
700515	Iasi	IS	(null)	(null)	(null)	(null)	(null)	(null)
705300	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705310	Focsani	VN	(null)	(null)	(null)	(null)	(null)	(null)
705800	Suceava	SV	(null)	(null)	(null)	(null)	(null)	(null)
706400	Birlad	VS	(null)	(null)	(null)	(null)	(null)	(null)
706510	Vaslui	VS	(null)	(null)	(null)	(null)	(null)	(null)

Figura 8.24. Codurile poştale care nu figurează în adresa niciunui client

Care este situația încasării facturii 1114?

Revenim la exemple din paragrafele 6.7.2 şi 6.7.4 (de la funcțiile-agregat SUM şi MAX). Spuneam, la momentele respective, că soluțiile formulate sunt valabile numai pentru facturile cu măcar o tranşă de încasare. Beneficiind de avantajele joncțiunii externe, se poate redacta o soluție ameliorată care va funcționa în orice situație. Am ales, pentru început, factura 1114 pentru că aceasta nu are nicio tranşă de încasare. Dacă vom joncționa extern la stânga rezultatul joncțiunii dintre LINIIFACT şi PRODUSE cu tabela ÎNCASFACT:

```
SELECT *
FROM (liniifact lf
	INNER JOIN produse p ON lf.CodPr = p.CodPr)
	LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE lf.NrFact = 1114
```

se va obține rezultatul din figura 8.25.

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	CODPR_1	DENPR	UM	GRUPA	PROCTVA	CODINC	NRFACT_1	TRANSA
1114	1	2	70	1070	2	Produs 2	kg	Bere	0,09	(null)	(null)	(null)
1114	2	4	30	1705	4	Produs 4	l	Dulciuri	0,09	(null)	(null)	(null)
1114	3	5	700	7064	5	Produs 5	buc	Tigari	0,19	(null)	(null)	(null)

Figura 8.25. Folosirea joncțiunii externe pentru aflarea situației încasărli facturii 1114

Răspunsul la întrebare se află destul de simplu (vezi cele două valori în figura 8.26):

```
SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS "Facturat",
	SUM(COALESCE(Transa,0)) AS "Incasat"
FROM (liniifact lf
	INNER JOIN produse p ON lf.CodPr = p.CodPr)
	LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
```

WHERE lf.NrFact = 1114

Facturat	Incasat
6021706,5	0

Figura 8.26. Valorile (corecte) facturate și încasate pentru factura 1114

Soluția nu este însă corectă pentru toate facturile, după cum vom vedea imediat.

Care este situația încasării facturii 1117?

Spre deosebire de 1114, această factură prezintă trei tranșe de încasare, rezultatul joncțiunii externe (identic cu al joncțiunii interne):

```
SELECT *
FROM (liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr )
        LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE lf.NrFact = 1117
```

fiind cel din figura 8.27.

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	CODPR_1	DENPR	UM	GRUPA	PROCTVA	CODINC	NRFACT_1	TRANSA
1117	2	1	100	950	1	Produs 1	buc	Tigari	0,19	1236	1117	9754
1117	1	2	100	1000	2	Produs 2	kg	Bere	0,09	1236	1117	9754
1117	2	1	100	950	1	Produs 1	buc	Tigari	0,19	1237	1117	9754
1117	1	2	100	1000	2	Produs 2	kg	Bere	0,09	1237	1117	9754
1117	2	1	100	950	1	Produs 1	buc	Tigari	0,19	1239	1117	3696
1117	1	2	100	1000	2	Produs 2	kg	Bere	0,09	1239	1117	3696

Figura 8.27. Folosirea joncțiunii externe pentru a afla situația încasării facturii 1117

După cum am convenit spre finalul paragrafului 6.7.4, pentru obținerea valorii facturate trebuie împărțită suma totală la numărul de tranșe de încasări (repetarea se producea din cauza joncțiunii). Atunci când nu există nicio tranșă, împărțirea trebuie să se facă la 1, astfel încât expresia valorii totale a facturii va fi: *SUM (Cantitate * PretUnit * (1 + ProcTVA))/COUNT(DISTINCT VALUE (i.CodInc, 1))*. În schimb, totalul tranșelor încasate (0 dacă nu există nicio tranșă) trebuie împărțit, pentru fiecare factură, la numărul de linii din factură. De aici expresia: *SUM (VALUE (Transa, 0))/MAX (lf.Linie)*. Prin urmare, soluția valabilă pentru toate facturile pare a fi:

```
SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/
        COUNT(DISTINCT I.CodInc) AS "Facturat",
        SUM(Transa)/MAX(LF.Linie) AS "Incasat"
FROM (liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr)
        LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE lf.NrFact = 1117
```

Cele două valori sunt identice cu cele din figura 6.61 (paragraful 6.7.2). Supriză, însă! Dacă revenim cu soluţia aceasta la factura 1114, ne alegem cu un mesaj de eroare (vezi figura 8.28). Ruşinea pe care am păţit-o e cauzată de faptul că, dacă factura nu are nicio tranşă de încasare, numitorul expresiei *SUM(Cantitate * PretUnit * (1 + ProcTVA))/COUNT(DISTINCT i.CodInc)* este zero. Din fericire, putem rezolva problema folosindu-ne de funcţia COALESCE în două locuri:

- pentru conversia eventualelor valori NULL ale CodInc în zero, astfel încât COUNT să le ia în considerare;
- pentru conversia eventualelor valori NULL ale atributului Tranşa în zero, astfel încât *SUM(COALESCE(Transa,0))/MAX(LF.Linie)* să returneze zero.

Ecce rezolvarea!

```
SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/
        COUNT(DISTINCT COALESCE(i.CodInc,0)) AS "Facturat",
        SUM(COALESCE(Transa,0))/MAX(lf.Linie) AS "Incasat"
FROM (liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr)
        LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE lf.NrFact = 1114
```

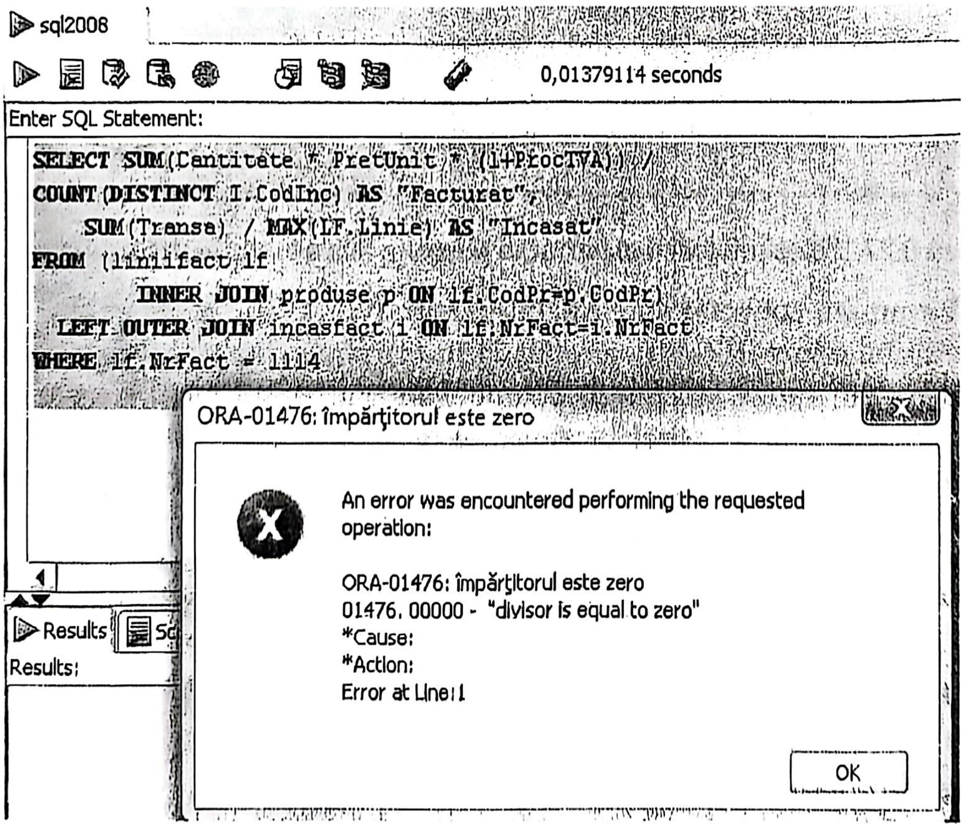


Figura 8.28. Eroarea la execuţia interogării pentru factura 1114

Care sunt valorile facturate şi încasate ale fiecărei facturi emise în luna august 2007?

Şi acesta este un caz rezolvat incomplet în paragraful 7.1 (figura 7.7). Beneficiind de experienţa exemplului anterior, formulăm soluţia:

```
SELECT lf.NrFact,
        SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT COALESCE(i.CodInc,0)) AS "Facturat",
        SUM(COALESCE(Transa,0))/MAX(lf.Linie) AS "Incasat"
FROM (facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr)
        LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 8
GROUP BY lf.NrFact
ORDER BY 1
```

Rezultatul pare a fi corect (vezi figura 8.29). Toate soluțiile din acest paragraf, cu excepția celei de mai sus, sunt acceptate de cele patru dialecte. Aceasta, însă, este funcțională doar în Oracle și PostgreSQL[8].

NRFACT	Facturat	Incasat
1111	4346037,5	53996
1112	125516	125516
1113	106275	106275
1114	6021706,5	0
1115	151237,5	0
1116	126712,5	0
1117	222050	23204
1118	201975	201975
1119	5774498,5	0
1120	97664	7315
1121	4737838	0
2111	4442959,5	0
2112	153442	0
2113	127530	0
2115	110907,5	0
2116	136849,5	0
2117	287855	0
2118	179565	0
2119	5819668	0
2121	4741271,5	0

Figura 8.29. Valorile facturate și încasate ale fiecărei facturi

Care au fost sporurile de noapte acordate angajaților pe lunile mai și iunie 2007?

Situația obținută se referă la două luni. Există însă angajați care nu au acest spor pe mai sau pe iunie sau chiar pe nicio lună (din cele două). Cu interogarea:

8. În DB2 și SQL Server, cele două funcții EXTRACT trebuie înlocuite cu YEAR și MONTH.

```
SELECT An, Luna, personal2.Marca, NumePren, SporNoapte
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE An = 2007 AND Luna IN (5,6)
ORDER BY NumePren, An, Luna
```

rezultatul arată ca în figura 8.30.

AN	LUNA	MARCA	NUMEPREN	SPORNOAPTE
2007	5	1	ANGAJAT 1	0
2007	6	1	ANGAJAT 1	0
2007	5	10	ANGAJAT 10	0
2007	6	10	ANGAJAT 10	12
2007	5	2	ANGAJAT 2	45
2007	6	2	ANGAJAT 2	0
2007	5	3	ANGAJAT 3	0
2007	6	4	ANGAJAT 4	15
2007	6	5	ANGAJAT 5	15

Figura 8.30. Sporurile de noapte pe mai şi iunie – varianta 1 de afişare

Pe noi ne interesează, însă, o formă mai elegantă de prezentare, ca în figura 8.31.

MARCA	NUMEPREN	SPORNOAPTE_MAI	SPORNOAPTE_IUNIE
1	ANGAJAT 1	0	0
10	ANGAJAT 10	0	12
2	ANGAJAT 2	45	0
3	ANGAJAT 3	0	(null)
4	ANGAJAT 4	(null)	15
5	ANGAJAT 5	(null)	15
6	ANGAJAT 6	(null)	(null)
7	ANGAJAT 7	(null)	(null)
8	ANGAJAT 8	(null)	(null)
9	ANGAJAT 9	(null)	(null)

Figura 8.31. Sporurile de noapte pe mai şi iunie – rezultatul dorit

Dacă am joncţiona intern tabelele PERSONAL2 cu SPORURI pentru luna mai 2007, rezultatul ar conţine atâtea linii câţi angajaţi au primit spor de noapte în această lună. Pot fi, dar, la fel de bine, nu pot fi toţi. Pentru a forţa includerea în rezultat a tuturor angajaţilor, apelăm la joncţiunea externă la stânga, obţinând tabela din figura 8.32.

```
SELECT personal2.Marca, NumePren,
        SporNoapte AS SporNoapte_Mai
FROM personal2
        LEFT OUTER JOIN sporuri ON personal2.Marca = sporuri.Marca
                AND An = 2007 AND Luna = 5
ORDER BY NumePren
```

MARCA	NUMEPREN	SPORNOAPTE_MAI
1	ANGAJAT 1	0
10	ANGAJAT 10	0
2	ANGAJAT 2	45
3	ANGAJAT 3	0
4	ANGAJAT 4	(null)
5	ANGAJAT 5	(null)
6	ANGAJAT 6	(null)
7	ANGAJAT 7	(null)
8	ANGAJAT 8	(null)
9	ANGAJAT 9	(null)

Figura 8.32. Sporurile de noapte pe luna mai (pentru toți angajații)

Persoanele care nu erau angajate în această perioadă prezintă valori NULL pentru atributele An și Lună. Pentru afișarea pe coloane separate a sporurilor de noapte pe lunile mai și iunie (ca în figura 8.31) sunt necesare două joncțiuni externe ale tabelei PERSONAL2 cu două instanțe ale tabelei SPORURI (instanțe care vor avea aliasurile s1 și s2).

```
SELECT personal2.Marca, NumePren,
        s1.SporNoapte AS SporNoapte_Mai,
        s2.SporNoapte AS SporNoapte_Iunie
FROM personal2
        LEFT OUTER JOIN sporuri s1 ON personal2.Marca = s1.Marca
        AND s1.An = 2007 AND s1.Luna = 5
        LEFT OUTER JOIN sporuri s2 ON personal2.Marca = s2.Marca
AND s2.An = 2007 AND s2.Luna = 6
ORDER BY NumePren
```

Elementul-cheie îl constituie prezența operatorului joncțiunii externe în dreptul atributelor An și Lună. Prin această se includ în rezultat și liniile din tabela PERSONAL2 care nu prezintă corespondență după atributul Marcă cu tabela SPORURI pentru cele două luni.

Să se obțină sporurile de noapte pentru al doilea trimestru al anului 2007, atât lunar, cât și cumulat.

Sunt necesare trei instanțe ale tabelei SPORURI (notate s1, s2 și s3), fraza SELECT devenind un pic mai amplă:

```
SELECT personal2.Marca, NumePren,
        COALESCE(s1.SporNoapte,0) AS SpNoapte_Aprilie,
        COALESCE(s2.SporNoapte,0) AS SpNoapte_Mai,
        COALESCE(s3.SporNoapte,0) AS SpNoapte_Iunie,
        COALESCE(s1.SporNoapte,0) + COALESCE (s2.SporNoapte,0)
                + COALESCE (s3.SporNoapte,0) AS Sp_Noapte_Trim2
```

```
FROM personal2
        LEFT OUTER JOIN sporuri s1 ON personal2.Marca = s1.Marca
                AND s1.An = 2007 AND s1.Luna = 4
        LEFT OUTER JOIN sporuri s2 ON personal2.Marca = s2.Marca
                AND s2.An = 2007 AND s2.Luna = 5
        LEFT OUTER JOIN sporuri s3 ON personal2.Marca = s3.Marca
                AND s3.An = 2007 AND s3.Luna = 6
ORDER BY NumePren
```

Rezultatul este asemănător celui din figura 8.33.

MARCA	NUMEPREN	SPNOAPTE_APRILIE	SPNOAPTE_MAI	SPNOAPTE_IUNIE	SP_NOAPTE_TRIM2
1	ANGAJAT 1	0	0	0	0
10	ANGAJAT 10	0	0	12	12
2	ANGAJAT 2	45	45	0	90
3	ANGAJAT 3	156	0	0	156
4	ANGAJAT 4	0	0	15	15
5	ANGAJAT 5	0	0	15	15
6	ANGAJAT 6	0	0	0	0
7	ANGAJAT 7	0	0	0	0
8	ANGAJAT 8	0	0	0	0
9	ANGAJAT 9	0	0	0	0

Figura 8.33. Sporurile de noapte pe trimestrul al II-lea, pe luni şi cumulat

În PostgreSQL, pentru asemenea gen de problemă putem folosi o funcţie „proprietară" care „orizontalizează verticalităţile"[9] - CROSSTAB:

```
SELECT *
FROM crosstab(
        'SELECT Marca, Luna, SporNoapte
        FROM sporuri
        WHERE An = 2007 AND luna BETWEEN 4 AND 6
        ORDER BY 1,2,3'
                )
                AS SpNoapte_Orizontalizat (
                        Marca NUMERIC,
                        SpNoapte_Apr NUMERIC,
                        SpNoapte_Mai NUMERIC,
                        SpNoapte_Iun NUMERIC
                                        )
```

Argumentul funcţiei este o interogare SQL în care specificăm faptul că liniile care ne interesează în rezultat sunt în tabela SPORURI cu valorile anului 2007, luna

9. Titulatura este preluată din *Fiola de SQL* nr. 4 din „îndepărtatul" *NetReport* nr. 95, din luna august 2000, şi inspirată din fotbal (vezi explicaţiile din fiolă).

trebuie să fie cuprinsă între 4 și 6[10], iar rezultatul tabelar să aibă numele SPNOAPTE_ORIZONTALIZAT, titulatura și tipologia coloanelor acestuia fiind precizate între parantezele ce urmează clauzei AS. Ceea ce se obține în urma execuției acestei interogări PostgreSQL seamănă leit cu figura 8.34.

marca numeric	spnoapte_apr numeric	spnoapte_mai numeric	spnoapte_iun numeric
1	0.00	0.00	0.00
2	45.00	45.00	0.00
3	156.00	0.00	
4	15.00		
5	15.00		
10	0.00	12.00	

Figura 8.34. Funcția PostgreSQL CROSSTAB()

Astfel, în PostgreSQL putem obține o listă similară celei din figura 8.33 joncționând extern la dreapta rezultatul funcției CROSSTAB cu tabela PERSONAL2:

```
SELECT p2.Marca, p2.NumePren,
        COALESCE(SpNoapte_Apr,0.00) AS SpNoapte_Apr,
        COALESCE(SpNoapte_Mai,0.00) AS SpNoapte_Mai,
        COALESCE(SpNoapte_Iun,0.00) AS SpNoapte_Iun,
        COALESCE(SpNoapte_Apr,0.00) + COALESCE(SpNoapte_Mai,0) +
                COALESCE(SpNoapte_Iun,0) AS Sp_Noapte_Trim2
FROM crosstab(
        'SELECT Marca, Luna, SporNoapte
        FROM sporuri
        WHERE An = 2007 AND luna BETWEEN 4 AND 6
        ORDER BY 1,2,3')
                AS sp_tab(
                        Marca NUMERIC,
                        SpNoapte_Apr NUMERIC,
                        SpNoapte_Mai NUMERIC,
                        SpNoapte_Iun NUMERIC
                        )
        RIGHT OUTER JOIN personal2 p2 ON sp_tab.Marca = p2.Marca
 ORDER BY 2
```

Să se afişeze, pentru toţi clienţii; valoarea vânzărilor pe lunile august şi septembrie 2007 şi totalul vânzărilor pe cele două luni; în plus, pe acelaşi raport să se afişeze totalul vânzărilor pe fiecare lună (pentru toţi clienţii) şi pe ambele.

10. Pentru detalii privind sintaxa și modul de funcționare al funcției, vezi documentația PostgreSQL 8.3, la pagina *http://www.postgresql.org/docs/8.3/interactive/tablefunc.html*.

Dacă interogarea anterioară a fost destul de amplă, ce să mai spunem despre aceasta?:

```
SELECT DenCl,
        ROUND(COALESCE(SUM(lf1.Cantitate * lf1.PretUnit *
                (1 + p1.ProcTVA)),0)) AS Vanz_Aug2007,
        ROUND(COALESCE(SUM(lf2.Cantitate * lf2.PretUnit *
                (1 + p2.ProcTVA)),0)) AS Vanz_Sep2007,
        ROUND(SUM(lf1.Cantitate * lf1.PretUnit *
                (1 + p1.ProcTVA))) + ROUND(COALESCE(SUM(lf2.Cantitate *
                        lf2.PretUnit * (1 + p2.ProcTVA)),0))
                AS Vanz_AugSep2007
FROM clienti c
        LEFT OUTER JOIN
                (
                facturi f1 INNER JOIN liniifact lf1 ON f1.NrFact = lf1.NrFact
                        AND EXTRACT (YEAR FROM DataFact) = 2007
                        AND EXTRACT (MONTH FROM DataFact) = 8
        INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
                ) ON c.CodCl = f1.CodCl
        LEFT OUTER JOIN
        (
        facturi f2 INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                AND EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
        INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
        ) ON c.CodCl = f2.CodCl
        GROUP BY DenCl
                UNION
SELECT 'T O T A L ',
        ROUND(SUM(lf1.Cantitate * lf1.PretUnit *
                        (1 + p1.ProcTVA))) AS Vanz_Aug2007,
        ROUND(COALESCE(SUM(lf2.Cantitate * lf2.PretUnit *
                        (1 + p2.ProcTVA)),0)) AS Vanz_Sep2007,
        ROUND(SUM(lf1.Cantitate * lf1.PretUnit * (1+p1.ProcTVA))) +
                ROUND(COALESCE(SUM(lf2.Cantitate *
                        lf2.PretUnit * (1 + p2.ProcTVA)),0)) AS Vanz_AugSep2007
FROM clienti c
        LEFT OUTER JOIN
        (
        facturi f1 INNER JOIN liniifact lf1 ON f1.NrFact = lf1.NrFact
```

```
            AND EXTRACT (YEAR FROM DataFact) = 2007
            AND EXTRACT (MONTH FROM DataFact) = 8
    INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
    ) ON c.CodCl = f1.CodCl
    LEFT OUTER JOIN
    (
    facturi f2 INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
            AND EXTRACT (YEAR FROM DataFact) = 2007
    AND EXTRACT (MONTH FROM DataFact) = 9
    INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
    ) ON c.CodCl = f2.CodCl
ORDER BY DenCl
```

Rândul de total are tipicul său şi de aceea interogarea conţine două fraze SELECT conectate prin UNION. În prima, cea corespunzătoare rândurilor „curente" din raport se joncţionează extern la stânga tabela CLIENŢI cu două instanţe ale joncţiunii FACTURI-LINIIFACT-PRODUSE; fiecare din cele două instanţe corespunde uneia din cele două luni calendaristice. Probabil că rezultatul din figura 8.35 merită efortul. Sintaxa este Oracle/PostgreSQL, dar trecerea la varianta DB2/SQL Server nu mai ridică, de acum, probleme deosebite, singurele vizate de schimbări fiind funcţiile EXTRACT şi ROUND.

DENCL	VINZ_AUG2007	VINZ_SEP2007	VINZ_AUGSEP2007
Client 1 SRL	80322008	85361879	165683887
Client 2 SA	233805	255060	488865
Client 3 SRL	46376666	46557344	92934010
Client 4	9479110	0	9479110
Client 5 SRL	557916	613768	1171684
Client 6 SA	6021707	0	6021707
Client 7 SRL	263562	273699	537261
T O T A L	143254773	133061750	276316523

Figura 8.35. Totalizări pe orizontală şi verticală

8.5. Structuri de control alternative

Deşi sună atrăgător, în acest paragraf nu vom vorbi despre programarea propriu-zisă, ci despre evaluarea unei condiţii (predicat) şi, în funcţie de variantele testate, returnarea unor valori. Ca multe alte generalizări, nici propoziţia anterioară nu spune mare lucru. Mai precis (cu 7%!), în clauza SELECT (cu precădere) a unei interogări vom introduce uneori o serie de condiţii, cu ajutorul unei structuri CASE sau similare, prin care valoarea extrasă în rezultat să depindă de rezultatul evaluării expresiei pentru linia respectivă.

În SQL, structura alternativă înseamnă CASE, automat structură „alternativă multiplă[11]". Unii vor fi nemulţumiţi de inexistenţa structurii simple - IF sau IIF (IF imediat). Dacă ne gândim la păstrarea simplităţii sintaxei SQL, existenţa doar a structurii CASE este acceptabilă.

Determinaţi sexul fiecărui pacient după CNP-ul său.

După cum am discutat şi în paragraful 6.5, folosind funcţii de extragere a unor porţiuni dintr-un şir de caractere, putem afla câteva informaţii pe baza CNP-ului. Dacă prima cifră dintr-un cod numeric personal este 1 sau 5, atunci sexul este masculin, iar dacă este 2 sau 6 este feminin.

O structură de tip CASE este delimitată de cuvintele CASE şi END. Între acestea, sunt descrise diferite variante de testare (WHEN...) şi valoarea returnată la îndeplinirea condiţiei testate (THEN...). Nefiind prea mult de testat, CASE-ul nostru seamănă mai degrabă cu un IF.

```
SELECT NumePacient, CNP,
        CASE
                WHEN SUBSTR(CAST (CNP AS CHAR(13)), 1, 1)
                        IN ('1', '5') THEN 'Barbatesc'
                ELSE 'Femeiesc'
        END AS Sex
FROM pacienti
ORDER BY NumePacient[12]
```

Atunci când toate variantele de test iau în calcul valori punctuale dintr-un acelaşi atribut (sau expresie), atributul (sau expresia) poate fi plasat imediat după CASE:

```
SELECT NumePacient, CNP,
        CASE SUBSTR(CAST (CNP AS CHAR(13)), 1, 1)
                WHEN '1'THEN 'Barbatesc'
                WHEN '5'THEN 'Barbatesc'
                ELSE 'Femeiesc'
        END AS Sex
FROM pacienti
ORDER BY NumePacient
```

Ambele variante obţin rezultatul din figura 8.36.

---

11. În româneşte, „structură alternativă multiplă" sună rău, ca să nu spun că-i pleonasm în toată regula, atâta vreme cât alternativa presupune alegerea unei variante din două posibile. Pentru cei încruntaţi, aş vrea să arunc pisica către alte barbarisme, informatice şi noninformatice, gen *setare, customizare, focusare, target* etc.

12. Varianta SQL Server a acestei interogări este: *SELECT numepacient, CNP, CASE WHEN SUBSTRING(STR(CNP,13), 1, 1) IN ('1', '5') THEN 'Barbatesc' ELSE 'Femeiesc' END AS Sex FROM pacienti ORDER BY numepacient*, adică se schimbă SUBSTR cu SUBSTRING, iar numele atributelor trebuie să fie scrise cu aceleaşi combinaţii litere mari-litere mici ca în CREATE TABLE.

NUMEPACIENT	CNP	SEX
Bagdasar Adela	2601002250611	Femeiesc
Buzatu Corneliu	1650512370514	Barbatesc
Cazan Ana Maria	2690202200345	Femeiesc
Ionascu Ionelia	6001122390199	Femeiesc
Popescu Maria-Mirabela	2721231300888	Femeiesc
Spineanu Marius	5010101380625	Barbatesc
Stroe Mihaela	2601228390834	Femeiesc
Stroescu Mihaela Oana	6020719120545	Femeiesc

Figura 8.36. Determinarea sexului din CNP-ul pacienţilor

Din păcate, în ultima variantă după WHEN se pot scrie numai valori, nefiind permise (în Oracle, PostgreSQL sau SQL Server) liste sau intervale. Astfel, următoarele două variante sunt eronate (vezi figura 8.37):

```
SELECT NumePacient, CNP,
        CASE SUBSTR(CAST (CNP AS CHAR(13)), 1, 1)
                WHEN '1' OR '5' THEN 'Barbatesc'
                ELSE 'Femeiesc'
        END AS Sex
FROM pacienti
```

şi

```
SELECT NumePacient, CNP, CASE SUBSTR(CAST (CNP AS CHAR(13)), 1, 1)
WHEN IN ('1', '5') THEN 'Barbatesc' ELSE 'Femeiesc' END AS Sex
FROM pacienti
```

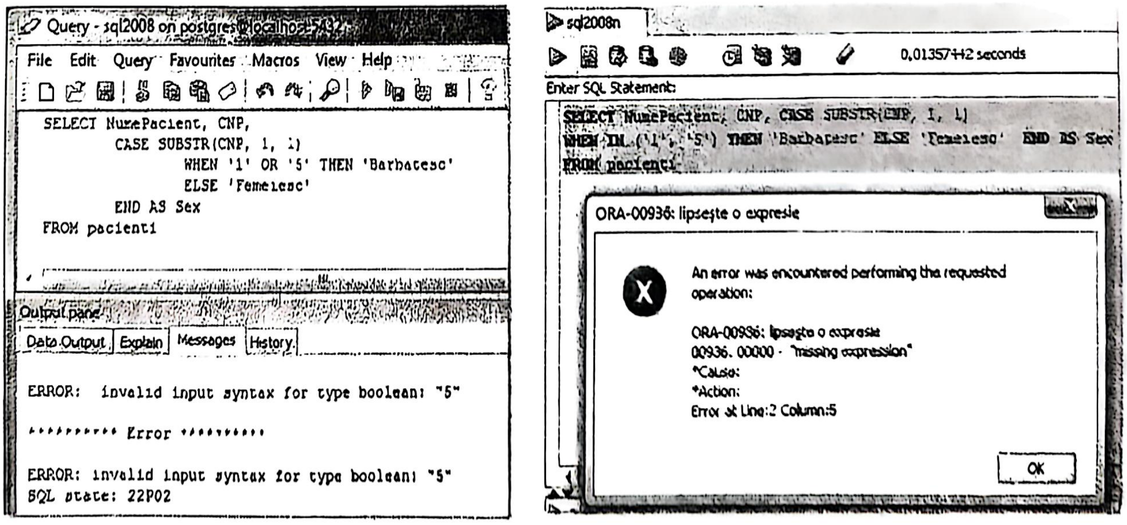


Figura 8.37. Limite ale CASE-ului

Chiar şi celor două variante corecte le putem reproşa că atunci când prima cifră a CNP-ului este eronată (nu e niciuna dintre valorile 1, 2, 5 şi 6), sexul returnat va fi Femeiesc, din cauza ramurii ELSE. Drept urmare variantele corecte sunt:

```
SELECT NumePacient, CNP,
```

```
        CASE
            WHEN SUBSTR(CAST (CNP AS CHAR(13)), 1, 1) IN ('1', '5') THEN
        'Barbatesc'
            WHEN SUBSTR(CAST (CNP AS CHAR(13)), 1, 1) IN ('2', '6') THEN 'Femeiesc'
        ELSE 'Sex necunoscut!'
        END AS Sex
FROM pacienti
ORDER BY NumePacient
```

şi

```
SELECT NumePacient, CNP,
        CASE SUBSTR(CAST (CNP AS CHAR(13)), 1, 1)
            WHEN '1' THEN 'Barbatesc'
            WHEN '2' THEN 'Femeiesc'
            WHEN '5' THEN 'Barbatesc'
            WHEN '6' THEN 'Femeiesc'
        ELSE 'Sex necunoscut!'
        END AS Sex
FROM pacienti
ORDER BY NumePacient
```

Putem privi acum funcţiile de conversie ale valorilor NULL ca pe nişte „scurtături" ale structurilor de tip CASE. Spre exemplu, interogarea pentru *aflarea totalului sporurilor pentru fiecare angajat pe luna iulie 2007* (vezi paragraful 8.3, figura 8.19) poate fi rescrisă astfel:

```
SELECT sporuri.Marca, NumePren, Compart,
        CASE WHEN SporVechime IS NULL THEN 0 ELSE SporVechime END
        + CASE WHEN SporNoapte IS NULL THEN 0 ELSE SporNoapte END
        + CASE WHEN SporCD IS NULL THEN 0 ELSE SporCD END
        + CASE WHEN AlteSpor IS NULL THEN 0 ELSE AlteSpor END
        AS TotalSporuri
FROM personal2 INNER JOIN sporuri
        ON personal2.Marca = sporuri.Marca
WHERE An = 2007 AND Luna = 7
```

Chiar dacă efortul intelectual este constant, lungimea acestei versiuni, altminteri corectă, este sensibil mai mare, deci avantajul este de partea funcţiilor de conversie a NULL-ităţilor. În plus, ajungem, ca în zilele bune, la consensul sintaxei celor patru dialecte SQL.

Să se afişeze facturile din luna septembrie 2007 în ordinea alfabetică a observaţiilor.

Aceeaşi problemă din DB2, PostgreSQL (pre-8.3) şi MS SQL Server (cauzată de inexistenţa opţiunilor NULLS FIRST şi NULLS LAST), pe care o soluţionam

printr-o funcție COALESCE, poate fi rezolvată printr-un CASE în clauza ORDER BY, după cum urmează:

```
SELECT *
FROM facturi
WHERE DataFact >= '2007-09-01' AND DataFact <= '2007-09-30'
ORDER BY CASE WHEN Obs IS NULL THEN ' ' ELSE Obs END – NULLS FIRST
```

și:

```
SELECT * FROM facturi
WHERE DataFact >= '2007-09-01' AND DataFact <= '2007-09-30'
ORDER BY CASE WHEN Obs IS NULL THEN 'zzz' ELSE Obs END – NULLS LAST
```

Aflați pentru toți pacienții data nașterii, cunoscându-le CNP-urile.

În același celebru paragraf 6.5 am afișat, pe baza CNP-ului, data nașterii pacienților, dar numai pentru cei născuți fie înainte de 2000, fie după 2000 (figurile 6.33 și 6.35). Cu acea ocazie dibuiam o ciudățenie a lucrului cu intervale în Oracle (figura 6.34). Beneficiind de structurile de control de tip CASE și de experiența exemplului de acum două pagini și jumătate, în Oracle putem rezolva problema afișării corecte a datei nașterii, „dintr-o suflare", întrucât cei cu cifra „sexului" 1 sau 2 sunt născuți înainte de 2000, iar cei cu cifra 5 sau 6 după 2000 (vezi figura 8.38).

```
SELECT NumePacient, cnp,
        CASE
                WHEN SUBSTR(cnp, 1, 1) IN ('1','2')
                        THEN ADD_MONTHS(TO_DATE(SUBSTR(cnp, 2, 6),
                                'YYMMDD'), -1200)
                ELSE TO_DATE(SUBSTR(cnp, 2, 6), 'YYMMDD')
        END AS Data_Nasterii_CNP
FROM pacienti
ORDER BY NumePacient
```

NUMEPACIENT	CNP	DATA_NASTERII_CNP
Bagdasar Adela	2601002250611	02-10-1960
Buzatu Corneliu	1650512370514	12-05-1965
Cazan Ana Maria	2690202200345	02-02-1969
Ionascu Ionelia	6001122390199	22-11-2000
Popescu Maria-Mirabela	2721231300888	31-12-1972
Spineanu Marius	5010101380625	01-01-2001
Stroe Mihaela	2601228390834	28-12-1960
Stroescu Mihaela Oana	6020719120545	19-07-2002

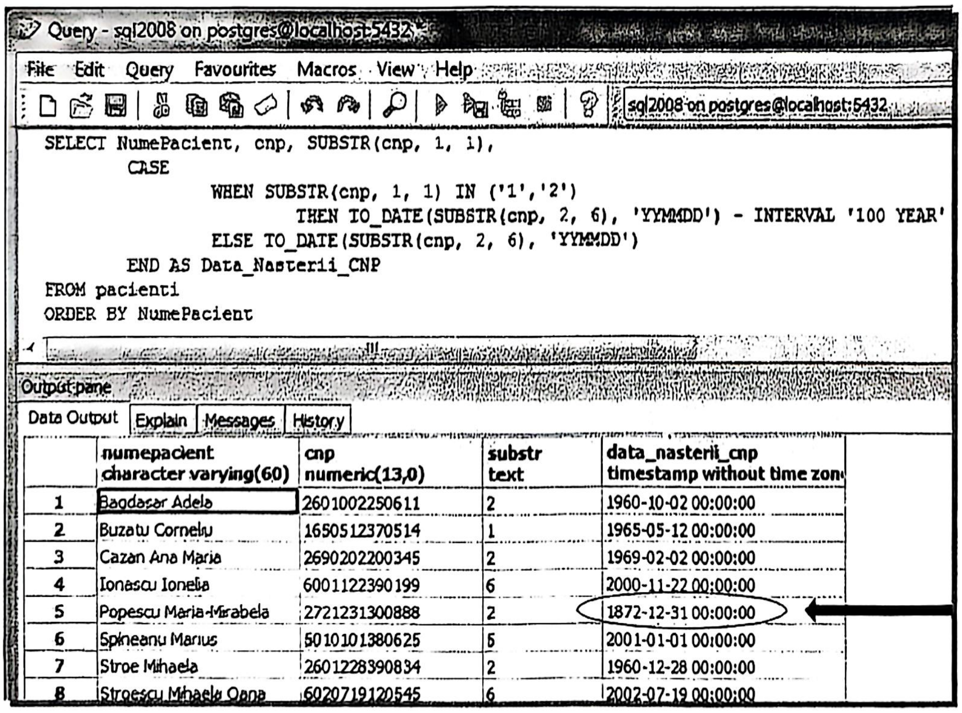
Figura 8.38. Obținerea datei nașterii din CNP-ul pacienților

Insistând asupra erorilor/ciudățeniilor din Oracle, am ajuns să fim nedrepți cu „ghidușiile" din alte dialecte. Încercăm să mai dregem busuiocul, prezentându-vă

un mic intermezzo PostgreSQL. Interogarea de mai sus nu funcţionează din cauza funcţiei ADD_MONTHS care este proprietară Oracle. Schimbăm ADD_MONTHS cu scăderea unui interval de 100 de ani:

```
SELECT NumePacient, cnp, SUBSTR(CAST (cnp AS CHAR(13)), 1, 1),
       CASE
       WHEN SUBSTR(CAST (cnp AS CHAR(13)), 1, 1) IN ('1','2')
              THEN TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6), 'YYMMDD') –
                     INTERVAL '100 YEAR'
       ELSE TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6), 'YYMMDD')
       END AS Data_Nasterii_CNP
FROM pacienti
ORDER BY NumePacient
```

se obţine rezultatul din figura 8.39. Ceea ce intrigă cel mai mult este data de pe linia 5, care este, nici mai mult, nici mai puţin, 31 decembrie 1872! Cu această ocazie aflăm că în PostgreSQL, atunci când anul dintr-o dată nu cuprinde şi secolul[13], dacă anul este înainte de anii '70 atunci va fi „interpretat" ca făcând parte din secolul 21 (2001-2069), iar dacă e după '70 e considerat ca făcând parte din secolul 20 (1970-2000).



	numepacient character varying(60)	cnp numeric(13,0)	substr text	data_nasterii_cnp timestamp without time zon
1	Bagdasar Adela	2601002250611	2	1960-10-02 00:00:00
2	Buzatu Corneliu	1650512370514	1	1965-05-12 00:00:00
3	Cazan Ana Maria	2690202200345	2	1969-02-02 00:00:00
4	Ionascu Ionelia	6001122390199	6	2000-11-22 00:00:00
5	Popescu Maria-Mirabela	2721231300888	2	1872-12-31 00:00:00
6	Spineanu Marius	5010101380625	5	2001-01-01 00:00:00
7	Stroe Mihaela	2601228390834	2	1960-12-28 00:00:00
8	Stroescu Mihaela Oana	6020719120545	6	2002-07-19 00:00:00

Figura 8.39. O dată calendaristică tulburătoare în PostgreSQL

Cunoscând aceste intimităţi, suntem în măsură să redactăm versiunea corectă:

```
SELECT NumePacient, cnp,
       TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6), 'YYMMDD'),
```

13. Să ne amintim că de la acest gen de reprezentări ale datelor calendaristice a pornit şi celebra *problemă a anului 2000* (Y2K).

```
        CASE
        WHEN SUBSTR(CAST (cnp AS CHAR(13)), 1, 1) IN ('1','2') THEN
                CASE
                WHEN CAST (SUBSTR(CAST (cnp AS CHAR(13)), 2, 2) AS NUMERIC)
                        < 70 THEN TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6),
                                'YYMMDD') - INTERVAL '100 YEAR'
                ELSE TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6), 'YYMMDD')
                END
        ELSE TO_DATE(SUBSTR(CAST (cnp AS CHAR(13)), 2, 6), 'YYMMDD')
        END AS Data_Nasterii_CNP
FROM pacienti
ORDER BY NumePacient
```

Adevărul este că ne-am complicat degeaba, pentru că putem renunţa la operaţiunile cu ani, secolul fiind specificat direct în şablon:

```
SELECT NumePacient, cnp, SUBSTR(CAST (cnp AS CHAR(13)), 1, 1),
        CASE
        WHEN SUBSTR(CAST (cnp AS CHAR(13)), 1, 1) IN ('1','2')
        THEN TO_DATE(('19' || SUBSTR(CAST (cnp AS CHAR(13)), 2, 6)),
        'YYYYMMDD')
        ELSE TO_DATE(('20' || SUBSTR(CAST (cnp AS CHAR(13)), 2, 6)),
        'YYYYMMDD')
        END AS Data_Nasterii_CNP
FROM pacienti
ORDER BY NumePacient
```

Interogarea funcţionează fără modificări şi în DB2. În SQL Server, dacă anul este specificat pe două poziţii, pentru intervalul 00-49 conversia se va face în 2000-2049, în timp ce anii din intervalul 50-99 vor fi consideraţi 1950-1999. Noi, însă, nu ne vom bizui pe acest lucru, ci vor ţine cont doar de cifra sexului (ca în interogarea PostgreSQL de mai sus):

```
SELECT numepacient, CNP,
        CASE
        WHEN SUBSTRING(STR(CNP,13), 1, 1) IN ('1','2') THEN
                CAST (('19' + SUBSTRING(STR(CNP,13),2,2)+'-'
                        + SUBSTRING(STR(CNP,13),4,2) +
                        '-' + SUBSTRING(STR(CNP,13),6,2)) AS SMALLDATETIME)
        ELSE
                CAST ('20' + SUBSTRING(STR(CNP,13),2,2) + '-'
                        + SUBSTRING(STR(CNP,13),4,2) +
                        '-' + SUBSTRING(STR(CNP,13),6,2) AS SMALLDATETIME)
        END AS Data_Nasterii_CNP
FROM pacienti
```

```
ORDER BY numepacient
```

Pare (şi chiar este) un efort nejustificat, însă putem rescrie şi interogarea pentru realizarea theta-joncţiunii (exemplul 20 din finalul paragrafului 4.4.4), cea cu afişarea numelui medicului care era de gardă în momentul sosirii fiecărui pacient în triaj. Echivalent soluţiei propuse în paragraful 5.7 (rezultatul este cel din figura 5.41), se poate redacta următoarea interogare SQL (sintaxă Oracle):

```
SELECT DataOra_Examinare, NumePacient, NumeDoctor
FROM triaj
        INNER JOIN garzi ON DataOra_Examinare =
                    CASE
                    WHEN DataOra_Examinare BETWEEN Inceput_Garda
                            AND Sfarsit_Garda THEN DataOra_Examinare
                    ELSE DataOra_Examinare + INTERVAL '1' DAY
                    END
        INNER JOIN doctori ON garzi.IdDoctor = doctori.IdDoctor
        INNER JOIN pacienti ON triaj.IdPacient = pacienti.IdPacient
ORDER BY 1
```

De data aceasta, theta-joncţiunea este, de fapt, echijoncţiune. Egalitatea testată este cea dintre Data_Ora_Examinare şi rezultatul furnizat de CASE. Sintaxa DB2 solicită schimbarea constantei interval din *INTERVAL '1' DAY* în *1 DAY*, iar în PostgreSQL forma aceleiaşi constante este *INTERVAL '1 DAY'*. În SQL Server, toate numele atributelor din cele patru tabele se scriu cu litere mici, iar expresia-interval *INTERVAL '1' DAY* se înlocuieşte cu *DATEADD(DAY, 1, data-ora_examinare)*.

Să se calculeze procentul de discount acordat fiecărei facturi din luna septembrie 2007, în situaţia în care acesta ar fi acordat pe trei tranşe de valori (totale) ale facturilor, după cum urmează:

- sub 15000 RON - 0%;
- între 15000 şi 199999 - 3%;
- între 199999 şi 399999 - 5%;
- între 400000 şi 600000 - 6%;
- peste 600000 - 7%.

Problema se rezolvă aplicând un CASE pentru fiecare linie (care se referă la o factură) a rezultatului. Ţinând seama de faptul că parcurgerea fiecărei ramuri de test este secvenţială, predicatul din clauza WHEN nu trebuie să conţină un interval, ci o simplă valoare:

```
SELECT f.NrFact, DataFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValFact,
        CASE
                WHEN TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0) < 150000
                        THEN 0
                WHEN TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0) < 200000
```

```
                THEN 0.03
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0) < 400000
                THEN 0.05
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1+ProcTVA)),0) < 600000
                THEN 0.06
        ELSE 0.07
        END * 100 || '%' AS ProcentDiscount
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
        EXTRACT(MONTH FROM DataFact) = 9
GROUP BY f.NrFact, DataFact
ORDER BY 1
```

Rezultatul (Oracle) este cel din figura 8.40, interogarea funcționând fără migrene și în PostgreSQL.

NRFACT	DATAFACT	VALFACT	PROCENTDISCOUNT
3111	01-09-2007	4442959	7%
3112	01-09-2007	153442	3%
3113	02-09-2007	127530	0%
3115	02-09-2007	110907	0%
3116	10-09-2007	136849	0%
3117	10-09-2007	287855	5%
3118	17-09-2007	179565	3%

Figura 8.40. Procentul de discount acordat pentru fiecare factură din septembrie 2007

Mai ciudat se comportă DB2-ul. Dacă dorim să afișăm procentul sub formă de număr:

```
SELECT f.NrFact, DataFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValFact,
        CASE
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) < 150000
                THEN 0
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) < 200000
                THEN 0.03
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) < 400000
                THEN 0.05
            WHEN TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) < 600000
                THEN 0.06
        ELSE 0.07
        END * 100 AS ProcentDiscount
```

```
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE YEAR (DataFact) = 2007 AND MONTH (DataFact) = 9
GROUP BY f.NrFact, DataFact
ORDER BY 1
```

rezultatul este corect (vezi stânga figurii 8.41). Dacă, însă, concatenăm rezultatul CASE-ului cu simbolul % pentru a obţine o listă ca în figura 8.40:

NRFACT	DATAFACT	VALFACT	PROCENTDISCOUNT
3111	Sep 1, 2007	4,442,959.0000	7.00
3112	Sep 1, 2007	153,442.0000	3.00
3113	Sep 2, 2007	127,530.0000	0.00
3115	Sep 2, 2007	110,907.0000	0.00
3116	Sep 10, 2007	136,849.0000	0.00
3117	Sep 10, 2007	287,855.0000	5.00
3118	Sep 17, 2007	179,565.0000	3.00

NRFACT	DATAFACT	VALFACT	PROCENTDISCOUNT
3111	Sep 1, 2007	4,442,959.0000	0000000000%
3112	Sep 1, 2007	153,442.0000	0000800000%
3113	Sep 2, 2007	127,530.0000	0000000000%
3115	Sep 2, 2007	110,907.0000	0000000000%
3116	Sep 10, 2007	136,849.0000	0000000000%
3117	Sep 10, 2007	287,855.0000	0000000000%
3118	Sep 17, 2007	179,565.0000	0000000000%

Figura 8.41. Nazuri DB2

```
SELECT f.NrFact, DataFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValFact,
        CAST ((CASE
        ...
        END * 100) AS CHAR(10)) || '%' AS ProcentDiscount
FROM
        ...
```

DB2-ul devine năzuros - vezi dreapta figurii 8.41.

Câţi clienţi sunt din localitatea Iaşi şi câţi din afara Iaşului?

Această problemă ne este de folos în a exemplifica folosirea unui CASE la grupare, sintaxa fiind valabilă în toate cele patru servere BD:

```
SELECT CASE Loc
                WHEN 'Iasi' THEN 'Din Iasi'
                ELSE 'Din afara Iasului'
        END AS Pozitionare,
        COUNT(*) AS NrClienti
FROM clienti INNER JOIN coduri_postale
        ON clienti.CodPost = coduri_postale.CodPost
GROUP BY CASE Loc WHEN 'Iasi' THEN 'Din Iasi'
        ELSE 'Din afara Iasului' END
```

POZITIONARE	NRCLIENTI
Din Iasi	2
Din afara Iasului	5

Figura 8.42. Numărul clienţilor ieşeni şi al celor din afara Ieşilor

Câţi angajaţi au primit, pe luna iulie 2007, spor pentru condiţii deosebite şi câţi nu?

Prin clauza WHERE sunt filtrate înregistrările din SPORURI, astfel încât vor fi luate în considerare numai datele din iulie 2007, iar gruparea se realizează printr-un CASE aplicat asupra valorii SporCD (vezi figura 8.43), sintaxă acceptată de toate cele patru dialecte SQL:

```
SELECT CASE
                WHEN SporCD > 0 THEN 'Au spor CD'
                ELSE 'Nu au spor CD'
        END AS "Situatie",
        COUNT(*) AS Nr
FROM sporuri
WHERE An = 2007 AND Luna = 7
GROUP BY CASE WHEN SporCD > 0
        THEN 'Au spor CD' ELSE 'Nu au spor CD' END
```

Situatie	NR
Nu au spor CD	8
Au spor CD	2

Figura 8.43. Numărul angajaţilor cu şi fără spor pentru condiţii deosebite în iulie 2007

Scadenţa fiecărei facturi emise este de 20 de zile. Dacă însă data-limită cade într-o sâmbătă sau duminică, atunci scadenţa se mută în lunea următoare. Care este, în aceste condiţii, data scadentă a fiecărei facturi emise în luna septembrie 2007?

Bănuim că trebuie să aplicăm un CASE expresiei care ne returnează ziua săptămânii în care cade data scadentă. Din păcate, funcţia ce returnează ziua din săptămână este nestandardizată. Să începem cu soluţia Oracle:

```
SELECT NrFact,
        DataFact AS "Data Emiterii",
        DataFact + INTERVAL '20' DAY AS "Data Scadenta",
        TO_CHAR(DataFact + INTERVAL '20' DAY, 'DAY') AS "Zi scadenta",
        CASE
                WHEN RTRIM(TO_CHAR( DataFact + INTERVAL '20' DAY,
                        'DAY')) IN ('SATURDAY', 'SÂMBĂTĂ')
                THEN DataFact + INTERVAL '22' DAY
        ELSE
                CASE
                        WHEN RTRIM(TO_CHAR(
                                DataFact + INTERVAL '20' DAY, 'DAY')) IN
                                        ('SUNDAY', 'DUMINICĂ')
                        THEN DataFact + INTERVAL '21' DAY
                ELSE DataFact + INTERVAL '20' DAY
```

```
            END
        END AS "Data scad.corectata",
        TO_CHAR(
            CASE
                WHEN RTRIM(TO_CHAR(DataFact + INTERVAL '20' DAY,
                        'DAY')) IN ('SATURDAY', 'SÂMBĂTĂ')
                THEN DataFact + INTERVAL '22' DAY
            ELSE
                CASE
                    WHEN RTRIM(TO_CHAR(DataFact +
                            INTERVAL '20' DAY, 'DAY')) IN
                            ('SUNDAY', 'DUMINICĂ')
                    THEN DataFact + INTERVAL '21' DAY
                    ELSE DataFact + INTERVAL '20' DAY
                    END
            END
        , 'DAY') AS "Zi scad.corectata"
FROM facturi
WHERE TO_CHAR(datafact, 'MM-YYYY') = '09-2007'
```

Aflarea zilei săptămânii presupune, după cum am văzut în paragraful 6.5, folosirea funcţiei TO_CHAR cu argumentul 'DAY'. Eventualele spaţii rezultate în urma conversiei sunt eliminate cu RTRIM(...). Întrucât este posibil ca sistemul de operare şi serverul Oracle să fie setate pe varianta engleză, dar, în unele cazuri (al meu, de exemplu) şi română am testat numele zilei de sâmbătă sau duminică atât cu titulatura în engleză ('SATURDAY'/'SUNDAY'), cât şi cu cea română ('SÂMBĂTĂ'/'DUMINICĂ'). Când scadenţa (data facturii + 20 de zile) cade într-o sâmbătă, o corectăm cu două zile (data facturii + 22), când cade într-o duminică, o corectăm cu o zi (data facturii + 21), iar când nu e nici sâmbătă, nici duminică, o păstrăm aşa (data facturii + 20). Lungimea exagerată a frazei SELECT e cauzată de repetarea secvenţei CASE necesară afişării denumirii zilei. Rezultatul este cel din figura 8.44.

NRFACT	Data Emiterii	Data Scadenta	Zi scadenta	Data scad.corectata	Zi scad.corectata
3111	01-09-2007	21-09-2007	VINERI	21-09-2007	VINERI
3112	01-09-2007	21-09-2007	VINERI	21-09-2007	VINERI
3113	02-09-2007	22-09-2007	SÂMBATA	24-09-2007	LUNI
3115	02-09-2007	22-09-2007	SÂMBATA	24-09-2007	LUNI
3116	10-09-2007	30-09-2007	DUMINICA	01-10-2007	LUNI
3117	10-09-2007	30-09-2007	DUMINICA	01-10-2007	LUNI
3118	17-09-2007	07-10-2007	DUMINICA	08-10-2007	LUNI

Figura 8.44. Scadenţa rectificată a facturilor

Secvenţa CASE este prea complicată din cauza prezenţei a două structuri CASE imbricate. Structura seamănă mai mult cu un IF tradiţional decât cu un CASE. Aşa

că pentru varianta PostgresSQL o simplificăm. În plus, pentru aflarea zilei din săptămână vom apela la EXTRACT (DOW...) descrisă în capitolul 6:

```
SELECT NrFact,
        DataFact AS "Data Emiterii",
        DataFact + INTERVAL '20 DAYS' AS "Data Scadenta",
        TO_CHAR(DataFact + INTERVAL '20 DAYS', 'DAY') AS "Zi scadenta",
        CASE EXTRACT (DOW FROM DataFact + INTERVAL '20 DAYS')
                WHEN 6 THEN DataFact + INTERVAL '22 DAYS' – sambata
                WHEN 0 THEN DataFact + INTERVAL '21 DAYS' – duminica
        ELSE DataFact + INTERVAL '20 DAYS' – luni-vineri
        END AS "Data scad.corectata",
        TO_CHAR (
                CASE EXTRACT (DOW FROM DataFact + INTERVAL '20 DAYS')
                        WHEN 6 THEN DataFact + INTERVAL '22 DAYS' – sambata
                        WHEN 0 THEN DataFact + INTERVAL '21 DAYS' – duminica
                        ELSE DataFact + INTERVAL '20 DAYS' – luni-vineri
                END
                , 'DAY') AS "Zi scad.corectata"
FROM FACTURI
WHERE TO_CHAR(datafact, 'MM-YYYY') = '09-2007'
```

Soluţia DB2 se bazează pe funcţia DAYOFWEEK, care întoarce 1, dacă *data-argument* se referă la duminică, 2 pentru luni, ..., 6 pentru sâmbătă. DAYNAME afişează numele zilei:

```
SELECT NrFact,
        DataFact AS "Data Emiterii",
        DataFact + 20 DAYS AS "Data Scadenta",
        DAYNAME(DataFact + 20 DAYS) AS "Zi scadenta",
        CASE DAYOFWEEK_ISO (DataFact + 20 DAYS)
                WHEN 6 THEN DataFact + 22 DAYS – sambata
                WHEN 7 THEN DataFact + 21 DAYS – duminica
                ELSE DataFact + 20 DAYS – luni-vineri
        END AS "Data scad.corectata",
        DAYNAME(CASE DAYOFWEEK_ISO (DataFact + 20 DAYS)
                WHEN 6 THEN DataFact + 22 DAYS – sambata
                WHEN 7 THEN DataFact + 21 DAYS – duminica
                ELSE DataFact + 20 DAYS – luni-vineri
                END) AS "Zi scad.corectata"
FROM FACTURI
WHERE MONTH(DataFact) = 9 AND YEAR(DataFact) = 2007
```

În fine, pentru echilibru, vom prezenta şi o variantă MS SQL Server:

```
SELECT NrFact,
        DataFact AS "Data Emiterii",
        DataFact + 20 AS "Data Scadenta",
        DATEPART(WEEKDAY, DataFact + 20) AS "Nr Zi scadenta",
        DATENAME(WEEKDAY, DataFact + 20) AS "Nume Zi scadenta",
        CASE DATEPART (WEEKDAY, DataFact + 20)
                WHEN 7 THEN DataFact + 22 – sambata
                WHEN 1 THEN DataFact + 21 – duminica
                ELSE DataFact + 20 – luni-vineri
        END AS "Data scad.corectata",
        DATENAME(WEEKDAY,
                        CASE DATEPART (WEEKDAY, DataFact + 20 )
                                WHEN 7 THEN DataFact + 22 – sambata
                                WHEN 1 THEN DataFact + 21 – duminica
                        ELSE DataFact + 20 – luni-vineri
                        END ) AS "Zi scad.corectata"
FROM facturi
WHERE DATEPART(MONTH,DataFact) = 9 AND DATEPART (YEAR,DataFact) = 2007
```

Care este valoarea vânzărilor din fiecare zi a săptămânii?

Acum cred că suntem în măsură să rezolvăm onorabil problema din paragraful 7.3. Ordinea dispunerii zilelor săptâmânii în rezultat ca în figura 7.20 se obţine apelând la o strucrură CASE în clauza ORDER BY. Iată o variantă comună Oracle/PostgreSQL:

```
SELECT TO_CHAR(DataFact, 'day') AS Zi_Saptamana,
        TRUNC(
                SUM(Cantitate * PretUnit * (1 + ProcTVA))
                ,0) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY TO_CHAR(DataFact, 'day')
        ORDER BY
        CASE
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('luni', 'monday') THEN 1
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('marţi', 'tuesday') THEN 2
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('miercuri', 'wednesday') THEN 3
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('joi', 'thursday') THEN 4
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('vineri', 'friday') THEN 5
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('sâmbătă', 'saturday') THEN 6
        WHEN RTRIM(TO_CHAR(DataFact, 'day')) IN ('duminică', 'sunday') THEN 7
```

```
        END
```

pe cea MS SQL Server:

```
SELECT DATENAME(WEEKDAY, DataFact) AS Zi_Saptamana,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DATENAME(WEEKDAY, DataFact)
ORDER BY
        CASE LOWER(DATENAME(WEEKDAY, DataFact))
        WHEN 'monday' THEN 1
        WHEN 'tuesday' THEN 2
        WHEN 'wednesday' THEN 3
        WHEN 'thursday' THEN 4
        WHEN 'friday' THEN 5
        WHEN 'saturday' THEN 6
        WHEN 'sunday' THEN 7
        END
```

și pe cea DB2:

```
SELECT DAYNAME(DataFact) AS Zi_Saptamana,
        TRUNC(
                SUM(Cantitate * PretUnit * (1 + ProcTVA))
                ,0) AS Vanzari_Zi_Sapt
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DAYNAME(DataFact)
ORDER BY
        CASE
        WHEN DAYNAME(DataFact) IN ('luni', 'monday') THEN 1
        WHEN DAYNAME(DataFact) IN ('marți', 'tuesday') THEN 2
        WHEN DAYNAME(DataFact) IN ('miercuri', 'wednesday') THEN 3
        WHEN DAYNAME(DataFact) IN ('joi', 'thursday') THEN 4
        WHEN DAYNAME(DataFact) IN ('vineri', 'friday') THEN 5
        WHEN DAYNAME(DataFact) IN ('sâmbătă', 'saturday') THEN 6
        WHEN DAYNAME(DataFact) IN ('duminică', 'sunday') THEN 7
        END
```

Să se afle numărul de produse vândute, pe următoarele intervale ale prețului unitar de vânzare: sub 1000 RON, între 1000 și 1999, 2000 și 2999 ș.a.m.d.

La momentul rezolvării acestei probleme (paragraful 7.3), cârteam la adresa figurii 7.28 în legătură cu modul de precizare a intervalelor. Cu ajutorul unei structuri CASE putem să facem rezultatul mai elegant (vezi figura 8.45):

```
SELECT CASE TRUNC(PretUnit,-3)
        WHEN 0 THEN '  < 1000'
        ELSE '>= ' || TO_CHAR(TRUNC(PretUnit,-3), '999999')
        END AS "Pret unitar ...",
        COUNT(*) AS "Nr.aparitii"
FROM liniifact
GROUP BY CASE TRUNC(PretUnit,-3)
        WHEN 0 THEN '  < 1000'
        ELSE '>= ' || TO_CHAR(TRUNC(PretUnit,-3), '999999')
        END
ORDER BY 1
```

Pret unitar ...	Nr.aparitii
< 1000	21
>= 1000	26
>= 6000	3
>= 7000	6

Figura 8.45. Ameliorarea rezultatului din figura 7.28

Sintaxa este comună - Oracle/PostgreSQL. Funcţia TO_CHAR, precum şi spaţiile de la primul interval - ' < 1000' -, ne asigură ordonarea corespunzătoare, indiferent de preţurile unitare din facturi. Acelaşi rezultat poate fi obţinut în SQL Server cu interogarea:

```
SELECT CASE ROUND(PretUnit,-3,1)
        WHEN 0 THEN '  < 1000'
        ELSE '>= ' + STR( ROUND(PretUnit,-3,1), 6)
        END AS "Pret unitar ...",
        COUNT(*) AS "Nr.aparitii"
FROM liniifact
GROUP BY CASE ROUND(PretUnit,-3,1)
        WHEN 0 THEN '  < 1000'
        ELSE '>= ' + STR( ROUND(PretUnit,-3,1), 6)
        END
ORDER BY 1
```

iar în DB2 astfel:

```
SELECT CASE TRUNC(PretUnit,-3)
        WHEN 0 THEN '  < 1000'
        ELSE '>= ' || CAST ( TRUNC(PretUnit,-3) AS CHAR(10))
        END AS "Pret unitar ...",
        COUNT(*) AS "Nr.aparitii"
FROM liniifact
```

```
GROUP BY
        CASE TRUNC(PretUnit,-3)
                WHEN 0 THEN '  <  1000'
        ELSE '>= ' || CAST ( TRUNC(PretUnit,-3) AS CHAR(10))
        END
ORDER BY 1
```

Să se afişeze în dreptul fiecărei facturi emise în luna august 2007: valoarea facturată, valoarea încasată, diferenţa de încasat, precum şi un text din care să reiasă dacă factura este fără nicio încasare, încasată parţial sau încasată total (şi cele trei valori).

Raportul dorit este o dezvoltare a celui din paragraful 8.5 (figura 8.29). Practic, se doreşte o situaţie cum este cea din figura 8.46. Fireşte, ultima coloană presupune folosirea unei structuri CASE:

```
SELECT lf.NrFact,
        TRUNC(
                SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                        COUNT(DISTINCT COALESCE(i.CodInc,0))
                ,0) AS "Facturat",
        TRUNC (
                SUM(COALESCE(Transa,0))/MAX(LF.Linie)
                ,0) AS "Incasat",
        TRUNC(
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) /
                        COUNT(DISTINCT COALESCE(i.CodInc,0))
                        -
                SUM(COALESCE(Transa,0)) / MAX(LF.Linie)
                ,0) AS "Diferenta",
        CASE
        WHEN  SUM(COALESCE(Transa,0)) / MAX(LF.Linie) = 0
                THEN ' Fara nicio incasare'
        WHEN SUM(Cantitate * PretUnit * (1+ProcTVA)) /
                COUNT(DISTINCT COALESCE(i.CodInc,0))
                >
                SUM(COALESCE(Transa,0))/MAX(LF.Linie)
                        THEN ' Incasata partial'
        ELSE ' *Incasata total*'
        END AS "Situatiune"
FROM (facturi f
        INNER JOIN liniifact lf ON f.nrfact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr)
```

```
        LEFT OUTER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 8
GROUP BY lf.NrFact
ORDER BY 1
```

NRFACT	Facturat	Incasat	Diferenta	Situatiune
1111	4346037	53996	4292041	Incasata partial
1112	125516	125516	0	*Incasata total*
1113	106275	106275	0	*Incasata total*
1114	6021706	0	6021706	Fara nici o incasare
1115	151237	0	151237	Fara nici o incasare
1116	126712	0	126712	Fara nici o incasare
1117	222050	23204	198846	Incasata partial
1118	201975	201975	0	*Incasata total*
1119	5774498	0	5774498	Fara nici o incasare
1120	97664	7315	90349	Incasata partial
1121	4737838	0	4737838	Fara nici o incasare
2111	4442959	0	4442959	Fara nici o incasare
2112	153442	0	153442	Fara nici o incasare
2113	127530	0	127530	Fara nici o incasare
2115	110907	0	110907	Fara nici o incasare
2116	136849	0	136849	Fara nici o incasare
2117	287855	0	287855	Fara nici o incasare
2118	179565	0	179565	Fara nici o incasare
2119	5819668	0	5819668	Fara nici o incasare
2121	4741271	0	4741271	Fara nici o incasare

Figura 8.46. Un nou raport privind încasarea facturilor din august 2007

În DB2 tot ce rămâne de făcut este să se înlocuiască EXTRACT cu funcţiile YEAR şi MONTH. La aceste modificări, în SQL Server mai trebuie adăugate cele care privesc sintaxa funcţiei TRUNCATE.

Oracle nu avea, până la versiunea 8, structura CASE... WHEN... În schimb, putea fi folosită funcţia DECODE, care are o logică similară. Interesant este că această funcţie apare şi în DB2, din versiunea 9.5. DECODE întoarce o valoare în funcţie de conţinutul unui argument:

```
DECODE (atribut,
        valoare_testată1, valoare_returnată1,
                valoare_testată2, valoare_returnată2,
                        ...
                valoare_returnatăN)
```

Valoare_retunatăN este echivalenta ELSE-ului dintr-o structură de tip CASE. Pentru a obţine sumara listă din figura 8.42, o altă soluţie Oracle/DB2 (post-9.5) ar fi:

```
SELECT DECODE (Loc,'Iasi', 'Din Iasi', 'Din afara Iasului') AS Pozitionare,
        COUNT(*) AS NrClienti
FROM clienti INNER JOIN coduri_postale
        ON clienti.CodPost = coduri_postale.CodPost
GROUP BY  DECODE (Loc,'Iasi', 'Din Iasi', 'Din afara Iasului')
```

Câţi angajaţi au primit, pe luna iulie 2007, spor pentru condiţii deosebite şi câţi nu?

Alăturăm „soluţiei CASE":

```
SELECT CASE
        WHEN SporCD > 0 THEN 'Au spor CD'
        ELSE 'Nu au spor CD'
        END AS Explicatii,
        COUNT(*) AS Nr
FROM sporuri
WHERE An = 2007 AND Luna = 7
GROUP BY CASE
        WHEN SporCD > 0 THEN 'Au spor CD'
        ELSE 'Nu au spor CD'
        END
```

o alta „proprietară" Oracle/DB2 bazată pe DECODE:

```
SELECT DECODE(SIGN( NVL(SporCD,0)-1), 1,
        'Au spor CD', 'Nu au spor CD')
                AS Pozitionare,
        COUNT(*) AS NrClienti
FROM SPORURI
WHERE An = 2007 AND Luna = 7
GROUP BY DECODE(SIGN( NVL(SporCD,0)-1), 1,
        'Au spor CD', 'Nu au spor CD')
```

Aveam nevoie de un mic truc pentru ca în DECODE să transformăm intervalele în listă. De aceea se scade 1 din *NVL(SporCD,0)* şi, dacă rezultatul are semnul +, înseamnă că s-a acordat sporul respectiv, iar în caz contrar, că nu s-a acordat.

Capitolul 9
Subconsultări

Una dintre cele mai importante facilităţi ale interogărilor SQL constă în folosirea rezultatului unei consultări ca argument într-o clauză WHERE, HAVING, FROM sau SELECT a unei alte consultări. Se poate ajunge la o structură destul de complicată de interogări şi subinterogări, dar şi rezultatele sunt pe măsură. Ar mai trebui spus că titlul acestui capitol putea fi *subconsultări necorelate,* deoarece, după cum vom vedea în capitolul următor, o categorie aparte de subconsultări este cea reprezentată de cele simplu sau dublu corelate.

9.1. Subconsultări în clauza WHERE. Operatorul IN

Operatorul cel mai utilizat în materie de subconsultări este IN pe care l-am întâlnit deja în paragraful 5.5 într-o cu totul altă ipostază - testarea încadrării valorii unui atribut într-o listă de constante. Pentru cele ce urmează, domeniul de acţiune al operatorului va fi o tabelă (ad-hoc) obţinută printr-o (sub)consultare.

9.1.1. Subconsultări în (sub)consultări în (sub)consultări în...

Revenim (iarăşi) la exemplul 19 din algebra relaţională (paragraful 4.4.4, figurile 4.24 şi 4.25): *Ce facturi au fost emise în aceeaşi zi cu factura 1120?* Anterior a fost formulată o soluţie bazată pe joncţiunea a două instanţe ale tabelei FACTURI. Iată însă o soluţie mai simplă bazată pe subconsultări:

```
SELECT *
FROM facturi
WHERE DataFact IN
        (SELECT DataFact
        FROM facturi
        WHERE NrFact = 1120)
```

Execuţia acestei interogări se derulează în doi timp. Mai întâi, se execută sub-consultarea *SELECT DataFact FROM FACTURI WHERE NrFact = 1120* obţinându-se o tabelă intermediară cu o singură linie şi o singură coloană (DataFact) (vezi partea stângă a figurii 9.1). La al doilea pas sunt selectate liniile tabelei FACTURI pentru care valoarea atributului DataFact este 7 august 2007.

DATAFACT
07-08-2007

NRFACT	DATAFACT	CODCL	OBS
1119	07-08-2007	1003	(null)
1120	07-08-2007	1001	(null)
1121	07-08-2007	1004	(null)
1122	07-08-2007	1005	(null)

Figura 9.1. Rezultatul subconsultării (stânga) şi cel final

În rezultat a fost inclusă şi factura de referinţă - 1120. Dacă se doreşte excluderea acesteia, fraza SELECT se modifică astfel:

```
SELECT *
FROM facturi
WHERE DataFact IN
        (SELECT DataFact
                FROM facturi
        WHERE NrFact = 1120)
        AND NrFact <> 1120
```

Ce facturi au fost emise în alte zile decât cea în care a fost emisă factura 1120?

Acest exemplu necesită folosirea operatorului de negaţie - NOT IN:

```
SELECT NrFact
FROM facturi
WHERE DataFact NOT IN
        (SELECT DataFact
        FROM facturi
        WHERE NrFact = 1120)
```

Care sunt clienţii cărora li s-au trimis facturi în aceeaşi zi în care a fost întocmită factura 1120?

```
SELECT DenCl
FROM clienti
WHERE CodCl IN
        (SELECT CodCl
        FROM facturi
        WHERE DataFact IN
                (SELECT DataFact
                FROM facturi
                WHERE NrFact = 1120)
        )
```

Figura 9.2 ilustrează modul ierarhic de execuţie a interogării cu trei niveluri de consultare (fraza principală, o subconsultare şi o subsubconsultare).

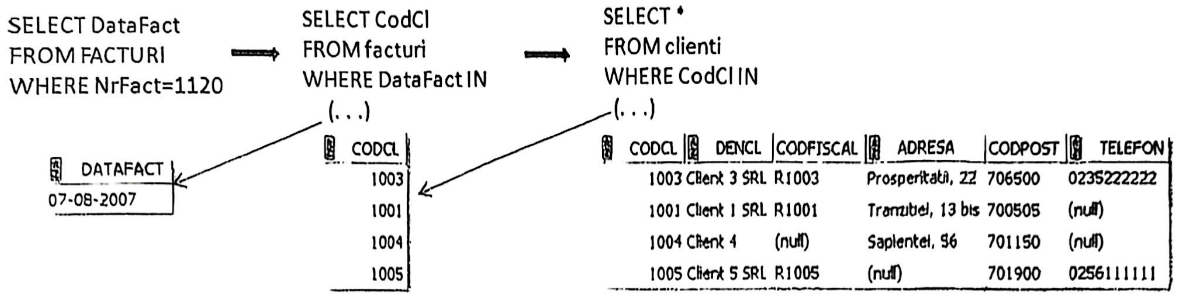


Figura 9.2. Mecanismul de execuţie a interogării cu două niveluri de subconsultare

La fel de corectă este şi varianta care foloseşte o joncţiune în locul unui nivel de subconsultare:

```
SELECT clienti.*
FROM clienti INNER JOIN facturi        ON clienti.CodCl = facturi.CodCl
WHERE DataFact IN
        (SELECT DataFact
        FROM facturi
        WHERE NrFact = 1120)
```

În ce judeţe s-a vândut produsul „Produs 2"?

Am ales acest exemplu pentru a „vântura", prin subconsultări, cât mai multe tabele ale bazei:

```
SELECT Judet
FROM judete
WHERE Jud IN
        (SELECT Jud
        FROM coduri_postale
        WHERE CodPost IN
                (SELECT CodPost
                FROM clienti
                WHERE CodCl IN
                        (SELECT CodCl
                        FROM facturi
                        WHERE NrFact IN
                                (SELECT NrFact
                                FROM liniifact
                                WHERE CodPr IN
                                        (SELECT CodPr
                                        FROM produse
                                        WHERE DenPr = 'Produs 2
                                        )
                                )
```

```
                    )
               )
          )
```

Revenim la tabela PERSONAL2 din figura 8.4: *Câţi subordonaţi direcţi are ANGAJAT 2?* La această problemă (la care răspunsul este 2) formulăm, pentru comparaţie, două soluţii. Soluţia bazată pe joncţiune este:

```
SELECT COUNT(*) AS NrSubordonati
FROM personal2 SUBORDONATI INNER JOIN personal2 SEFI
        ON SUBORDONATI.MarcaSef = SEFI.Marca
WHERE SEFI.NumePren = 'ANGAJAT 2'
```

Deşi trebuia să facem lucrul acesta mai demult, întrucât nu este o chestiune legată de subconsultări, să vedem cum se prezintă execuţia acestei interogări. Până acum, am mai joncţionat două instanţe ale unei tabele după un câmp comun. Noutatea acestei probleme ţine de faptul că tabela PERSONAL2, prin cele două atribute, Marca (cheia primară) şi MarcaSef (cheie străină) conţine structura ierarhică a firmei. Obţinem numele şefului direct al fiecărui angajat astfel:

```
SELECT SUBORDONATI.*,
        SEFI.Marca AS "Sefi.Marca",
        SEFI.NumePren AS "Sefi.NumePren",
        SEFI.SalTarifar AS "Sefi.SalTarifar"
FROM personal2 SUBORDONATI INNER JOIN personal2 SEFI
        ON SUBORDONATI.MarcaSef = SEFI.Marca
```

Din rezultat (figura 9.3) lipseşte angajatul cu marca 1, deoarece el este singurul fără superiori ierarhici. Pe fiecare linie a rezultatului se află, deci, un angajat împreună cu şeful său direct.

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR	Sefi.Marca	Sefi.NumePren	Sefi.SalTarifar
3	ANGAJAT 3	02-08-1962	MARKETING	1	1450	1	ANGAJAT 1	1600
10	ANGAJAT 10	29-01-1972	RESURSE UMANE	1	1370	1	ANGAJAT 1	1600
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450	1	ANGAJAT 1	1600
4	ANGAJAT 4	(null)	FINANCIAR	2	1380	2	ANGAJAT 2	1450
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420	2	ANGAJAT 2	1450
9	ANGAJAT 9	28-02-1976	MARKETING	3	1410	3	ANGAJAT 3	1450
8	ANGAJAT 8	31-12-1960	MARKETING	3	1290	3	ANGAJAT 3	1450
7	ANGAJAT 7	(null)	FINANCIAR	5	1280	5	ANGAJAT 5	1420
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350	5	ANGAJAT 5	1420

Figura 9.3. Şefii direcţi ai fiecărui angajat

A doua soluţie utilizează o subconsultare:

```
SELECT COUNT(Marca) AS NrSubordonati
FROM personal2
WHERE MarcaSef IN
        (SELECT Marca
        FROM personal2
        WHERE NumePren = 'ANGAJAT 2')
```

În paragraful 4.4.6 am prezentat, foarte pe scurt, un tip de joncțiune destul de discret – semijoncțiunea - prin care se extrag numai liniile dintr-o tabelă ce au corespondent (ca valoare a atributului de legătură) în a doua tabelă. Cu operatorul IN se pot formula lejer soluții ce corespund semijoncțiunii. Astfel, o tabelă precum cea din figura 4.29 se obține prin:

```
SELECT * FROM r1
WHERE C IN (SELECT C FROM r2)
```

Completăm discuția și cu exemplul „practic" luat pentru ilustrarea semijoncțiunii (exemplul 22 din paragraful 4.4.6): *Care sunt localitățile (codul poștal, denumirea și indicativul județului) în care există măcar un client?*

```
SELECT *
FROM coduri_postale
WHERE CodPost IN
        (SELECT CodPost
        FROM clienti)
```

Tot prin subconsultări putem realiza intersecția și diferența relațională. Raportându-ne la intersecția a două relații, R1 și R2, operațiunea se poate realiza în SQL și astfel:

```
SELECT *
FROM r1
WHERE (A,B,C) IN
        (SELECT C,D,E
        FROM r2)
```

Spre deosebire de interogările de până acum din acest capitol, care funcționează în toate cele patru servere BD, această variantă nu funcționează în SQL Server (dealtminteri, nici în Visual FoxPro sau Access), deoarece într-o subconsultare nu pot fi testate simultan trei valori (ale atributelor A, B și C), ci numai una. Astfel încât, pentru a-i înșela vigilența, se vor concatena cele trei atribute, dând „senzația" unuia singur:

```
SELECT *
FROM r1
WHERE CAST(A AS VARCHAR) + B + CAST (C AS VARCHAR) IN
        (SELECT CAST (C AS VARCHAR) + D + CAST (E AS VARCHAR)
        FROM r2)
```

Exemplul 17 din paragraful 4.4.4: *În ce zile s-au vândut și produsul cu denumirea „Produs 1" și cel cu denumirea „Produs 2"?*

```
SELECT DISTINCT DataFact
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
```

```
WHERE DenPr = 'Produs 1' AND DataFact IN
        (SELECT DataFact
        FROM produse
                INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
                INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
        WHERE DenPr = 'Produs 2')
```

Şi diferenţa relaţională poate fi realizată cu ajutorul subconsultărilor - operatorul NOT IN. Diferenţa relaţiilor R1 şi R2 din paragraful 4.3.3 (figura 4.5) este rezultatul SELECT-ului:

```
SELECT *
FROM r1
WHERE (A,B,C) NOT IN
        (SELECT C,D,E
        FROM r2)
```

Fireşte, în SGBD-urile în care „subiect" al subconsultărilor nu poate fi un tuplu ad-hoc (două sau mai multe atribute), cum este cazul SQL Server-ului, vom aplica „trucul" concatenării, ca şi la intersecţie.

Apelăm iarăşi la un exemplu din algebra relaţională (exemplu 18 din paragraful 4.4.4): *Ce clienţi au cumpărat şi „Produs 2" şi „Produs 3", dar nu au cumpărat „Produs 5"?*

```
SELECT DISTINCT DenCl
FROM produse
        INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
        INNER JOIN clienti ON facturi.CodCl = clienti.CodCl
WHERE DenPr = 'Produs 2' AND facturi.CodCl IN
        (SELECT CodCl
        FROM produse
                INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
                INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
        WHERE DenPr = 'Produs 3' AND facturi.CodCl NOT IN
                (SELECT CodCl
                FROM produse
                        INNER JOIN liniifact
                                ON produse.CodPr = liniifact.CodPr
                        INNER JOIN facturi
                                ON liniifact.NrFact=facturi.NrFact
                WHERE DenPr = 'Produs 5'
                )
        )
```

Până la apariția operatorilor LEFT OUTER JOIN, RIGHT OUTER JOIN și FULL OUTER JOIN, în numeroase SGBD-uri, joncțiunea externă era realizată prin reuniunea liniilor obținute din echijoncțiune cu liniile unei tabele (completate cu zerouri/spații pentru atributele celeilalte tabele) ce nu au corespondent în cealaltă. Iată o interogare ce joncționează extern la stânga relațiile R1 și R2 prin atributul C:

```
SELECT *
FROM r1 INNER JOIN r2 ON r1.C = r2.C
UNION
SELECT A,B,C, NULL, NULL, NULL
FROM r1
WHERE C NOT IN
        (SELECT C FROM r2)
```

Dacă rândurile din rezultat sunt identice cu cele obținute prin subconsultare, în Oracle titulatura coloanelor este cel puțin interesantă la execuția acestui SELECT (vezi figura 9.4).

A	B	QCSJ_C000000000300000	QCSJ_C000000000300001	D	E
20	XYZ	30	30	XXZ	40
30	XXZ	20	(null)	(null)	(null)
40	YYX	25	25	XYZ	30

Figura 9.4. Surprize, suprize în Oracle (alt episod)

În DB2 nu putem introduce direct NULL în lista coloanelor celui de-al doilea SELECT, ci vom apela la funcția CAST. În plus, pentru a asigura o titulatură dorită ar fi trebui să folosim clauza AS pentru fiecare coloană din ambele SELECT-uri.

```
SELECT *
FROM r1 INNER JOIN r2 ON r1.C = r2.C
UNION
SELECT A,B,C, CAST( NULL AS NUMERIC), CAST(NULL AS CHAR),
        CAST( NULL AS NUMERIC)
FROM r1
WHERE C NOT IN
        (SELECT C
        FROM r2)
```

Revenim (deși o să regretăm) la câteva exemple prezentate la joncțiunea externă pe care le rezolvăm prin noua „rețetă" - reuniune/valori vide.

Care sunt valorile facturate și încasate ale fiecărei facturi?

Ultima variantă de rezolvare este cea din partea finală a paragrafului 8.5. Soluția de mai jos (redactată în sintaxa Oracle/PostgreSQL) reunește facturile care au măcar o tranșă de încasare cu cele neîncasate deloc:

```
SELECT lf.NrFact,
        TRUNC(
```

```
            SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                    COUNT(DISTINCT COALESCE(i.CodInc,0))
            ,0) AS "Facturat",
        TRUNC (
            SUM(COALESCE(Transa,0))/MAX(lf.Linie)
            ,0) AS "Incasat",
        TRUNC(
            SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                    COUNT(DISTINCT COALESCE(i.CodInc,0))
            -
            SUM(COALESCE(Transa,0))/MAX(lf.Linie)
            ,0) AS "Diferenta",
            CASE
            WHEN SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                    COUNT(DISTINCT COALESCE(i.CodInc,0))
                    >
                    SUM(COALESCE(Transa,0))/MAX(lf.Linie)
                    THEN ' Incasata partial'
            ELSE ' *Incasata total*'
            END AS "Situatiune"
FROM facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN incasfact i ON lf.NrFact = i.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 8
GROUP BY lf.NrFact
        UNION
SELECT lf.NrFact,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))),
        0,
        TRUNC(SUM(Cantitate * PretUnit * (1 + ProcTVA))),
        ' Fara nici o incasare'
FROM facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
        AND EXTRACT (MONTH FROM DataFact) = 8
        AND F.NrFact NOT IN
                    (SELECT NrFact
```

```
                    FROM incasfact)
GROUP BY lf.NrFact
ORDER BY 1
```
[1]

Să se obţină sporurile de noapte pentru al doilea trimestru al anului 2007, atât lunar, cât şi cumulat.

Trebuie reunite persoanele care au sporul de noapte pe toate cele trei luni, cu persoanele care prezintă sporul numai pe câte două luni, cu persoanele cu sporul pe numai o singură lună, şi cu persoanele cărora nu li s-a calculat spor de noapte pe niciuna dintre cele trei luni.

Dacă la momentul formulării soluţiei anterioare afirmam că fraza SELECT este supraponderală, această soluţie este pur şi simplu pantagruelică. Cred că interogarea următoare demonstrează fără dubii că joncţiunea externă este o găselniţă grozav de inteligentă şi, pe de altă parte, că scrierea frazelor SQL poate deveni, pe alocuri, o treabă înfricoşătoare, dăunând, împreună cu excesul de sare, zahăr şi alcool, grav sănătăţii:

```
– primul SELECT este al celor care au sporuri de noapte
–           pe toate cele trei luni – 4,5 si 6
SELECT personal2.Marca, NumePren,
            s1.SporNoapte AS Spor_Noapte_Aprilie,
            s2.SporNoapte AS Spor_Noapte_Mai,
            s3.SporNoapte AS Spor_Noapte_Iunie,
            s1.SporNoapte + s2.SporNoapte + s3.SporNoapte
                    AS Spor_Noapte_Trim_II
FROM personal2
            INNER JOIN sporuri s1 ON personal2.Marca = s1.Marca
                    AND s1.An = 2007 AND 4 = s1.Luna
            INNER JOIN sporuri s2 ON personal2.Marca = s2.Marca
                    AND s2.An = 2007 AND 5 = s2.Luna
            INNER JOIN sporuri s3 ON personal2.Marca = s3.Marca
                    AND s3.An = 2007 AND 6 = s3.Luna
            UNION
– acest SELECT ii extrage pe cei care au avut spor de noapte
–           numai pe lunile 4 si 5
SELECT personal2.Marca, NumePren,
            s1.SporNoapte,
            s2.SporNoapte,
            0,
            s1.SporNoapte + s2.SporNoapte
```

1. Sintaxa DB2 şi SQL Server este diferită în privinţa funcţiei TRUNC şi necesită substituirea funcţiei EXTRACT prin YEAR şi MONTH.

```
FROM personal2
        INNER JOIN sporuri s1 ON personal2.Marca = s1.Marca
                AND s1.An = 2007 AND 4 = s1.Luna
        INNER JOIN sporuri s2 ON personal2.Marca = s2.Marca
                AND s2.An = 2007 AND 5 = s2.Luna
WHERE personal2.Marca NOT IN
                (SELECT Marca FROM sporuri WHERE An = 2007 AND Luna = 6)
        UNION
-- acest SELECT extrage pe cei care au avut spor de noapte
--      numai pe lunile 4 si 6
SELECT personal2.Marca, NumePren,
        s1.SporNoapte,
        0,
        s3.SporNoapte,
        s1.SporNoapte + s3.SporNoapte
FROM personal2
        INNER JOIN sporuri s1 ON personal2.Marca = s1.Marca
                AND s1.An = 2007 AND 4 = s1.Luna
        INNER JOIN sporuri s3 ON personal2.Marca = s3.Marca
                AND s3.An = 2007 AND 6 = s3.Luna
WHERE personal2.Marca NOT IN
                (SELECT Marca FROM sporuri WHERE An = 2007 AND Luna = 5)
        UNION
-- acest SELECT extrage pe cei care au avut spor de noapte
--      numai pe lunile 5 si 6
SELECT personal2.Marca, NumePren,
        0,
        s2.SporNoapte,
        s3.SporNoapte,
        s2.SporNoapte + s3.SporNoapte
FROM personal2
        INNER JOIN sporuri s2 ON personal2.Marca = s2.Marca
                AND s2.An = 2007 AND 5 = s2.Luna
        INNER JOIN sporuri s3 ON personal2.Marca = s3.Marca
                AND s3.An = 2007 AND 6 = s3.Luna
WHERE personal2.Marca NOT IN
                (SELECT Marca FROM sporuri WHERE An = 2007 AND Luna = 5)
        UNION
-- acest SELECT extrage pe cei care au avut spor de noapte numai pe luna 4
SELECT personal2.Marca, NumePren,
```

```
        s1.SporNoapte,
        0,
        0,
        s1.SporNoapte
FROM personal2 INNER JOIN sporuri s1 ON personal2.Marca = s1.Marca
                AND s1.An = 2007 AND 4 = s1.Luna
WHERE personal2.Marca NOT IN
                (SELECT Marca
                FROM sporuri
                WHERE An = 2007 AND Luna = 5)
                AND personal2.Marca NOT IN
                        (SELECT Marca FROM sporuri
                        WHERE An = 2007 AND Luna = 6)
        UNION
– acest SELECT extrage pe cei care au avut spor de noapte numai pe luna 5
SELECT personal2.Marca, NumePren,
        0,
        s2.SporNoapte,
        0,
        s2.SporNoapte
FROM personal2 INNER JOIN sporuri s2 ON personal2.Marca = s2.Marca
        AND s2.An = 2007 AND 5 = s2.Luna
WHERE personal2.Marca NOT IN
                (SELECT Marca FROM sporuri WHERE An = 2007 AND Luna = 4)
                AND personal2.Marca NOT IN
                        (SELECT Marca FROM sporuri
                        WHERE An = 2007 AND Luna = 6)
        UNION
– acest SELECT extrage pe cei care au avut spor de noapte numai pe luna 6
SELECT personal2.Marca, NumePren,
        0,
        0,
                s3.SporNoapte,
                s3.SporNoapte
    FROM personal2 INNER JOIN sporuri s3 ON personal2.Marca = s3.Marca
    AND s3.An = 2007 AND 6 = s3.Luna
            WHERE personal2.Marca NOT IN
                        (SELECT Marca FROM sporuri
                        WHERE An = 2007 AND Luna = 4)
            AND personal2.Marca NOT IN
```

```
                              (SELECT Marca FROM sporuri
                              WHERE An = 2007 AND Luna = 5)
     UNION
– acest SELECT extrage pe cei care nu au avut spor de noapte pe nici o luna
SELECT personal2.Marca, NumePren,
     0,
     0,
     0,
     0
FROM personal2
WHERE personal2.Marca NOT IN
                              (SELECT Marca  FROM sporuri
                              WHERE An = 2007 AND Luna = 4)
                         AND personal2.Marca NOT IN
                              (SELECT Marca FROM sporuri
                              WHERE An = 2007 AND Luna = 5)
                         AND personal2.Marca NOT IN
                              (SELECT Marca FROM sporuri
                              WHERE An = 2007 AND Luna = 6)
ORDER BY NumePren
```

O mângâiere (deşi e mai mult autoflagelare decât mângâiere) este că am scris cea mai lungă frază SELECT de până acum... Plus că sintaxa este comună celor patru servere BD.

9.1.2. Diviziunea prin subconsultări în clauza WHERE

Tot cu ajutorul operatorului IN (şi NOT IN) se poate aborda şi „problema" diviziunii relaţionale în SQL (parcă vă aud spunând: „Asta ne mai lipsea acum!"). Succesiunea paşilor ilustrată în algebra relaţională în figura 4.33 (paragraful 4.4.7) se realizează în SQL (dialectele DB2/PostgreSQL) astfel :

```
SELECT X
FROM rd1
     EXCEPT
SELECT DISTINCT rd1.X FROM rd1 CROSS JOIN rd2
WHERE (rd1.X, rd2.Y) NOT IN
     (SELECT X, Y FROM rd1)
```

În Oracle se înlocuieşte operatorul EXCEPT cu MINUS, iar SQL Server nu acceptă tupluri ad-hoc ca argumente ale operatorului IN, aşa că recurgem la soluţia concatenării.

```
SELECT DISTINCT X
FROM rd1
```

```
WHERE X NOT IN
        (
        SELECT DISTINCT rd1.X
        FROM rd1, rd2
        WHERE rd1.X + rd2.Y NOT IN
                (SELECT X + Y
                FROM rd1)
        )
```

sau

```
SELECT DISTINCT X
FROM rd1
WHERE X NOT IN (
                SELECT R1_1.X
                FROM (rd1 R1_1 INNER JOIN rd2 R2_1 ON 1=1)
                        LEFT OUTER JOIN rd1 R1_2 ON R1_1.X = R1_2.X
        AND R2_1.Y = R1_2.Y
                WHERE R1_2.Y IS NULL
                )
```

Această ultimă comandă poate fi lansată în oricare dintre cele patru dialecte. Pseudojoncțiunea internă este de fapt un produs cartezian, deoarece condiția de joncțiune este 1 = 1[2]. Subconsultarea extrage *icşii* care au „goluri", iar fraza SELECT principală efectuează scăderea lor din *icşii* tabelei RD1. Ultima interogare funcționează pe toate cele patru servere de baze de date.

Care sunt facturile ce conțin măcar produsele din factura 1117? (*Exemplul* 27 din paragraful 4.4.7). Soluția DB2/Oracle/PostgreSQL este:

```
SELECT NrFact
FROM liniifact
EXCEPT[3]
        SELECT DISTINCT rd1.NrFact
        FROM liniifact rd1, liniifact rd2
        WHERE rd2.NrFact = 1117 AND (rd1.NrFact, rd2.CodPr) NOT IN
                (SELECT NrFact, CodPr
                FROM liniifact)
```

iar cea SQL Server:

```
SELECT NrFact
FROM liniifact
EXCEPT
```

2. Era mai elegant să folosim *CROSS JOIN* în loc de ... *INNER JOIN... ON* 1 = 1.
3. MINUS (în loc de EXCEPT) în Oracle.

```
SELECT DISTINCT rd1.NrFact
FROM liniifact rd1, liniifact rd2
WHERE rd2.NrFact = 1117 AND
CAST (rd1.NrFact AS CHAR(8)) + CAST (rd2.CodPr AS CHAR(6))
        NOT IN
(SELECT CAST (NrFact AS CHAR(8)) + CAST (CodPr AS CHAR(6))
FROM liniifact)
```

Care sunt clienţii pentru care există cel puţin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007?

Acesta este *exemplul 23* din paragraful 4.4.7 (figura 4.31) pentru ilustrarea operatorului diviziune. Urmăm logica pe care tocmai am prezentat-o în Oracle/ PostgreSQL şi DB2 (dacă scoatem cuvintele DATE):

```
SELECT DenCl
FROM clienti
WHERE CodCl NOT IN
        (SELECT c.CodCl
        FROM clienti c CROSS JOIN facturi f
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
                AND (c.codcl, f.DataFact) NOT IN
                        (
                        SELECT CodCl, DataFact
                        FROM facturi f
                        WHERE DataFact BETWEEN DATE'2007-09-10'
        AND DATE'2007-09-30'
                        )
        )
```

Rămâne ca temă pentru acasă elaborarea variantei funcţionabile în MS SQL Server.

Ce produse au fost vândute tuturor clienţilor?

Acesta este exemplul 26 din paragraful 4.4.7 pentru ilustrarea folosirii operatorului diviziune. După scrutarea tabelelor PRODUSE, LINIIFACT, FACTURI şi CLIENŢI este evident că numai Produs 2 a fost vândut tuturor clienţilor. Cu toate acestea, urmând logica funcţională în DB2, PostgreSQL şi Oracle (înlocuind EXCEPT cu MINUS) pe care tocmai am prezentat-o:

```
SELECT DenPr
FROM produse
EXCEPT
        SELECT DISTINCT p1.DenPr
        FROM (produse p1
```

```
            INNER JOIN liniifact lf1 ON p1.CodPr = lf1.CodPr
            INNER JOIN facturi f1 ON lf1.NrFact = f1.NrFact)
                CROSS JOIN clienti rd2
        WHERE (p1.DenPr, rd2.CodCl) NOT IN
(SELECT DISTINCT DenPr, CodCl
FROM produse
INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
        INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
)
```

rezultatul conține două produse (vezi figura 9.5).

DENPR
Produs 2
Produs 6

Figura 9.5. Rezultat eronat al diviziunii (exemplul 26 din paragraful 4.4.7)

Intrusul este produsul 6. Acesta nu apare în nicio factură. De aceea, la joncțiunea cu LINIIFACT și FACTURI, nu va fi inclus în produsul cartezian și, astfel, nu va fi identificat ca lipsind din combinații cu toți clienții (CodCl) - în fapt, el nu este în combinație cu nici un client. Nefiind identificat ca absent în combinație cu toți clienții, va fi considerat „prezent".

Pentru a remedia situația, vom forța includerea în produsul cartezian a tuturor produselor, așa că vom apela la joncțiunea externă. În plus, clauza WHERE, care testează existența tuplurilor din produsul cartezian în „relația" ad-hoc cu atributele (DenPr, CodCl), va face apel la structura CASE:

```
SELECT DenPr
FROM produse
EXCEPT
        SELECT DISTINCT p1.DenPr
        FROM (produse p1 LEFT OUTER JOIN(
            liniifact lf1 INNER JOIN facturi f1 ON lf1.NrFact = f1.NrFact
                                    ) ON p1.CodPr = lf1.CodPr)
            CROSS JOIN clienti rd2
            WHERE (p1.DenPr,
                CASE WHEN lf1.CodPr IS NULL THEN 0
                ELSE rd2.CodCl
                END) NOT IN
                    (SELECT DISTINCT DenPr, CodCl
            FROM produse
                INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
                INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
            )
```

Iată şi varianta SQL Server:

```
SELECT DenPr
FROM produse
        EXCEPT
SELECT DISTINCT p1.DenPr
FROM (produse p1 LEFT OUTER JOIN (
        liniifact lf1 INNER JOIN facturi f1 ON lf1.NrFact = f1.NrFact
                ) ON p1.CodPr = lf1.CodPr)
                        CROSS JOIN clienti rd2
WHERE (p1.DenPr + CAST(CASE WHEN lf1.CodPr IS NULL THEN 0
                        ELSE rd2.CodCl END AS CHAR(6))) NOT IN
        (SELECT DISTINCT DenPr + CAST (CodCl AS CHAR(6))
        FROM produse
                INNER JOIN liniifact ON produse.CodPr = liniifact.CodPr
                INNER JOIN facturi ON liniifact.NrFact = facturi.NrFact
        )
```

9.1.3. Subconsultări şi NULL-ităţi

Am ameninţat în câteva rânduri că prezenţa valorilor NULL în liste-argument ale operatorului IN poate crea probleme. Să luăm un exemplu. Dacă până acum ne-au interesat facturile emise în aceeaşi zi cu factura 1120, acum ne frământă ideea de a afla *facturile cu aceleaşi observaţii ca ale facturii 1120*. După cum bănuiaţi, interogarea:

```
SELECT * FROM facturi
WHERE obs IN (SELECT obs FROM facturi WHERE NrFact = 1120)
```

nu conţine nici un rând, întrucât valoarea atributului Obs pentru factura-etalon este NULL. De aceea, trebuie reţinut, printre sfaturile părinteşti (pre- şi post-maritale) din SQL că, la folosirea subconsultărilor care ar extrage valori NULL, să facem conversia folosind funcţii COALESCE/NVL/VALUE:

```
SELECT * FROM facturi
WHERE COALESCE(Obs, ' ') IN
        (SELECT COALESCE(Obs, ' ')
        FROM facturi
        WHERE NrFact = 1120)
```

Deseori, o subconsultare extrage valori nenule combinate cu cele nule. De exemplu, în interogarea următoare:

```
SELECT *
FROM facturi
WHERE Obs IN
        (SELECT Obs
        FROM facturi
```

```
        WHERE NrFact = 1112 OR NrFact = 1111)
```

subconsultarea va extrage două linii, dintre care cea corespunzătoare facturii 1111 este NULL. Conform logicii predicatelor în care apar valori NULL, interogarea de mai sus afişează numai facturile care au observaţii nenule egale cu ale facturii 1112 (vezi figura 9.6).

NRFACT	DATAFACT	CODCL	OBS
3112	01-09-2007	1005	Probleme cu transportul
2112	14-08-2007	1005	Probleme cu transportul
1112	01-08-2007	1005	Probleme cu transportul

Figura 9.6. Valori nule extrase prin subconsultări

Dacă, însă, se foloseşte NOT IN:

```
SELECT *
FROM facturi
WHERE Obs NOT IN
        (SELECT Obs
        FROM facturi
        WHERE NrFact = 1112 OR NrFact = 1111)
```

rezultatul nu va conţine nici un rând. Spectrul NULL-ităţii este, pentru mulţi practicieni, suficient de stimulant pentru renunţarea la subconsultări în favoarea joncţiunii externe, însă, în situaţii precum aceasta, în care şi atributul testat (nu numai valorile extrase prin subconsultare) poate prezenta valori NULL, cel mai sănătos e să folosim funcţia COALESCE.

9.2. Subconsultări şi comparaţii în clauza WHERE

Până acum subconsultările au fost conectate la fraza SELECT superioară exclusiv prin operatorul IN. În continuare vom vedea că pentru (sub)interogările comparative, pot fi întrebuinţaţi ALL, SOME, ANY. Atunci când rezultatul unei subconsultări se concretizează într-o tabelă cu o singură coloană şi o singură linie (i se spune interogare scalară), corelarea poate fi făcută cu operatorii de comparaţie obişnuiţi: =, >, >=, <, <=. Vom ilustra această facilitate prin câteva exemple.

Ce facturi au fost emise în ziua în care a fost întocmită factura 1120?

Dacă în locul operatorului IN din paragraful anterior folosim egal, nu este nici o problemă:

```
SELECT NrFact
FROM facturi
WHERE DataFact =
            (SELECT DataFact
            FROM facturi
            WHERE NrFact = 1120)
```

În schimb, dacă în interogarea pentru aflarea *codurilor poştale (plus localitatea, indicativul judeţului) în care există măcar un client* schimbăm IN cu =:

SELECT * FROM coduri_postale

WHERE CodPost = (SELECT CodPost FROM clienti)

recepţionăm un mesaj de eroare în care ni se spune clar că subconsultarea conţine mai mult de o linie (vezi figura 9.7).

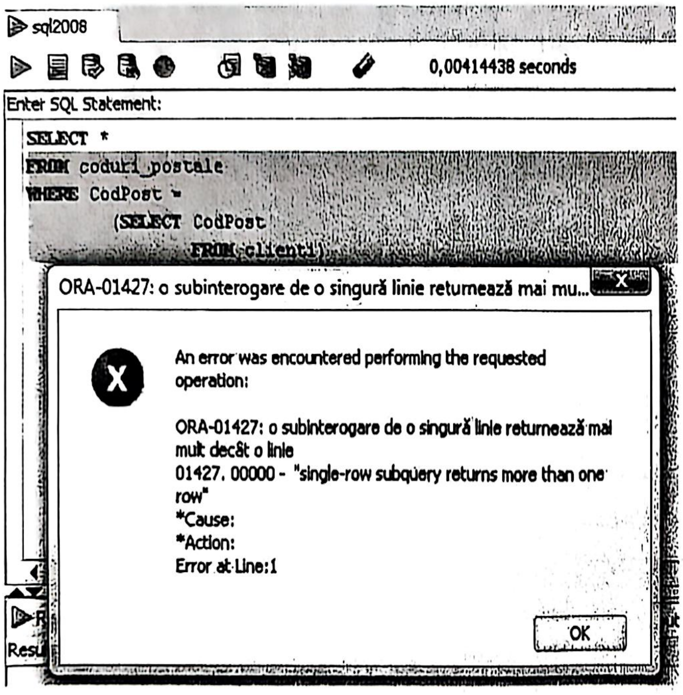


Figura 9.7. Folosirea eronată a semnului „=" pentru conexiunea cu subconsultarea

Care este cel mai mare preţ unitar la care a fost vândut un produs şi care sunt produsul, precum şi factura unde se înregistrează respectivul preţ?

Este una din problemele care ne-au dat de furcă în paragraful 6.7.4. Renunţând la *over*-doza de improvizaţie din acel paragraf, cu ajutorul subconsultărilor obţinem o soluţie foarte simplă:

SELECT NrFact, DenPr, PretUnit

FROM liniifact lf INNER JOIN produse p

ON lf.CodPr = p.CodPr

WHERE PretUnit = (SELECT MAX(PretUnit) FROM liniifact)

ORDER BY 1

De data aceasta rezultatul (vezi figura 9.8) este complet, în sensul că vor fi afişate toate facturile şi produsele în/pentru care se înregistrează preţul respectiv.

NRFACT	DENPR	PRETUNIT
1114	Produs 5	7064
1121	Produs 5	7064
2121	Produs 5	7064

Figura 9.8. Facturile şi produsele în/pentru care s-au înregistrat cel mai mare preţ

Care sunt cele mai mari două prețuri unitare de vânzare și produsele și facturile pentru care se înregistrează respectivele prețuri maxime?

Iată o variantă bazată pe subconsultări:

```
SELECT NrFact, DenPr, PretUnit
FROM liniifact lf INNER JOIN produse p
        ON lf.CodPr = p.CodPr
WHERE PretUnit >=
        (SELECT MAX(PretUnit)
        FROM liniifact
        WHERE PretUnit <
                        (SELECT MAX(PretUnit)
                        FROM liniifact
                        )
        )
ORDER BY PretUnit DESC, NrFact
```

Pentru a înțelege mecanismul acestei interogări, pornim de la SELECT-ul „cel mai de jos". *SELECT MAX(PrețUnit) FROM liniifact* extrage prețul unitar maxim din tabela LINIIFACT. Subconsultarea superioară, *(SELECT MAX(PretUnit) FROM liniifact WHERE PretUnit < (...ultima subconsultare...),* determină al doilea preț unitar din LINIIFACT. SELECT-ul principal afișează toate prețurile unitare mai mari sau egale cu penultimul. Figura 9.9 ilustrează acest mecanism.

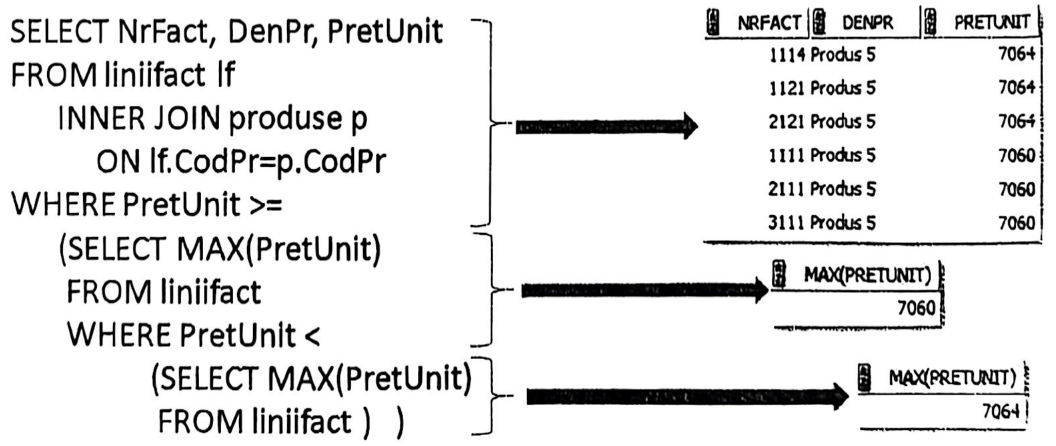


Figura 9.9. Cele mai mari două prețuri și facturile și produsele în/pentru care s-au înregistrat

Care sunt cele mai mari cinci prețuri unitare de vânzare și produsele și facturile în care apar cele cinci prețuri maxime?

Aici voiam, de fapt, să ajungem:

```
SELECT NrFact, DenPr, PretUnit
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE PretUnit >
        (SELECT MAX(PretUnit)
```

```
          FROM liniifact
          WHERE PretUnit <
          (SELECT MAX(PretUnit)
                    FROM liniifact
                    WHERE PretUnit <
          (SELECT MAX(PretUnit)
          FROM liniifact
          WHERE PretUnit <
(SELECT MAX(PretUnit)
FROM liniifact
WHERE PretUnit <
          (SELECT MAX(PretUnit)
          FROM liniifact
          WHERE PretUnit <
          (SELECT MAX(PretUnit)
                    FROM liniifact
          )
          )
)
          )
          )
          )
ORDER BY PretUnit DESC, NrFact
```

Logica este cea din soluţia anterioară, iar rezultatul este prezentat în figura 9.10. Celor tari de înger le sugerez să încerce cu primele 10, 20 ş.a.m.d. preţuri unitare.

NRFACT	DENPR	PRETUNIT
1114	Produs 5	7064
1121	Produs 5	7064
2121	Produs 5	7064
1111	Produs 5	7060
2111	Produs 5	7060
3111	Produs 5	7060
1119	Produs 5	6300
2119	Produs 5	6300
3119	Produs 5	6300
1114	Produs 4	1705
1119	Produs 4	1410
2119	Produs 4	1410
3119	Produs 4	1410

Figura 9.10. Cele mai mari cinci preţuri unitare

În PostgreSQL, o clauză de mare ajutor este LIMIT prin care se extrag primele *n* valori ale unei expresii dintr-un set de înregistrări. Astfel, soluția devine jenant de simplă:

```
SELECT NrFact, DenPr, PretUnit
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
WHERE PretUnit IN
        (SELECT DISTINCT PretUnit
        FROM liniifact
        ORDER BY PretUnit DESC
        LIMIT 5)
```

Bine, nu chiar jenant! Subconsultarea trebuie neapărat să folosească clauza DISTINCT ca să conțină cele mai mari cinci valori ale prețurilor unitare. La fel de simplă este clauza TOP din MS SQL Server (şi Visual FoxPro):

```
SELECT TOP 5 NrFact, DenPr, PretUnit
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
ORDER BY PretUnit DESC
```

şi clauza FETCH din DB2:

```
SELECT NrFact, DenPr, PretUnit
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
ORDER BY PretUnit DESC
FETCH FIRST 5 ROWS ONLY
```

După cum am văzut în figura 9.7, atunci când se foloseşte semnul „=" pentru a lega o subconsultare de (sub)consultarea „superioară", este necesar ca rezultatul subconsultării să fie scalar (o sigură linie şi o singură coloană). Lucrurile nu stau diferit nici cu ceilalți operatori de comparație (<, >, <=, >=, < > sau #). Dacă în cea mai mare parte a cazurilor de până acum se compara un atribut (sau rezultatul unei expresii/funcții) cu o valoarea scalară, prin clauzele ALL, SOME şi ANY se compară valoarea atributului/funcției/expresiei cu un set de tupluri (ansamblu de linii) extras printr-o subconsultare.

Care sunt produsele vândute la prețuri unitare superioare oricărui preț unitar la care a fost vândut „Produs 1"?

```
SELECT DISTINCT DenPr, PretUnit
FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
WHERE PretUnit > ALL
        (SELECT DISTINCT PretUnit
        FROM produse p INNER JOIN liniifact lf  ON p.CodPr = lf.CodPr
        WHERE DenPr = 'Produs 1')
ORDER BY DenPr, PretUnit DESC
```

Ca orice interogare pe două nivele, ostilitățile se derulează în doi paşi (vezi figura 9.11). Mai întâi se execută subconsultarea şi se obține o tabelă intermediară

în care se găsesc toate preţurile unitare la care a fost vândut, în decursul istoriei, Produs 1.

```
SELECT DISTINCT DenPr, PretUnit
FROM produse p
   INNER JOIN
     liniifact lf
       ON p.CodPr=lf.CodPr
WHERE PretUnit > ALL
       (SELECT DISTINCT PretUnit
        FROM produse p INNER JOIN
              liniifact lf ON p.CodPr=lf.CodPr
         WHERE DenPr ='Produs 1')
ORDER BY DenPr, PretUnit DESC
```

DENPR	PRETUNIT
Produs 2	1120
Produs 2	1100
Produs 2	1090
Produs 2	1070
Produs 2	1060
Produs 2	1030
Produs 4	1705
Produs 4	1410
Produs 5	7064
Produs 5	7060
Produs 5	6300

PRETUNIT
1000
930
950

Figura 9.11. Produsele vândute la preţuri mai mari decât toate preţurile produsului 1

Cum operatorul de conexiune a frazei SELECT principale cu subconsultarea este „> *ALL*", din joncţiunea tabelelor PRODUSE şi LINIIFACT vor fi extrase numai liniile care au valoarea atributului PreţUnit mai mare decât *toate* valorile din dreapta-jos figurii 9.11.

Care sunt produsele vândute la preţuri unitare superioare măcar unui preţ unitar al „Produsului 1"?

Este genul de situaţii în care se foloseşte SOME sau ANY (sunt echivalente).

```
SELECT DISTINCT DenPr, PretUnit
FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
WHERE PretUnit > ANY
        (SELECT DISTINCT PretUnit
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        WHERE DenPr = 'Produs 1')
ORDER BY DenPr, PretUnit DESC
```

Rezultatul este cel din figura 9.12; spre deosebire de ALL, atât ANY, cât şi SOME selectează liniile pentru care preţul unitar este mai mare decât *măcar una* dintre valorile obţinute prin subconsultare.

DENPR	PRETUNIT
Produs 1	1000
Produs 1	950
Produs 2	1120
Produs 2	1100
Produs 2	1090
Produs 2	1070
Produs 2	1050
Produs 2	1030
Produs 2	1000
Produs 2	975
Produs 4	1705
Produs 4	1410
Produs 5	7064
Produs 5	7060
Produs 5	6300

Figura 9.12. Produse cu cel puțin un preț unitar superior măcar unui preț al Produsului 1

Rețineți că operatorul „=ANY" este echivalent cu *IN*.

Câți alți angajați au salariul tarifar egal cu cel al ANGAJATului 2?

Soluția:

```
SELECT COUNT(*) - 1 AS Nr
FROM personal2
WHERE SalTarifar IN
        (SELECT SalTarifar
        FROM personal2
        WHERE NumePren = 'ANGAJAT 2')
```

este echivalentă cu:

```
SELECT COUNT(*) - 1 AS Nr
FROM personal2
WHERE SalTarifar = ANY
        (SELECT SalTarifar
        FROM personal2
        WHERE NumePren = 'ANGAJAT 2')
```

Fireşte, folosirea unuia dintre cei trei operatori nu este obligatorie atunci când subconsultarea conține o funcție-agregat (ce întoarce o valoare dintr-un ansamblu de tupluri) - MIN, MAX, COUNT, SUM, AVG.

Care este ultima factură întocmită (factura cea mai recentă) şi data în care a fost emisă?

Avem variante destule, chiar pe alese:

```
SELECT DataFact, NrFact AS UltimaFactura
FROM facturi
```

```
WHERE NrFact IN
        (SELECT MAX(NrFact)
        FROM facturi)
```
sau
```
SELECT DataFact, NrFact AS UltimaFactura
FROM facturi
WHERE NrFact =
        (SELECT MAX(NrFact)
        FROM facturi)
```
sau
```
SELECT DataFact, NrFact AS UltimaFactura
FROM facturi
WHERE NrFact = ANY
        (SELECT MAX(NrFact)
        FROM facturi)
```
sau
```
SELECT DataFact, NrFact AS UltimaFactura
FROM facturi
WHERE NrFact = ALL
        (SELECT MAX(NrFact)
        FROM facturi)
```
Fără funcţia MAX, interogarea ar avea forma:
```
SELECT DataFact, NrFact AS UltimaFactura
FROM facturi
WHERE NrFact >= ALL
        (SELECT NrFact
        FROM facturi)
```

9.3. Subconsultări în clauza HAVING

Predicatele incluse în clauza HAVING ale interogărilor din capitolele anterioare comparau o expresie cu o constantă. În continuare vom ataca problema includerii în clauza HAVING a subconsultărilor.

9.3.1. Comparaţii şi topuri la nivel de grupuri

Începem prin a reveni asupra unor probleme deja discutate, formulând câteva soluţii noi. Nu o să mai insistăm cu uşoarele modificări de sintaxă necesare portării comenzilor în toate cele patru dialecte SQL.

Care sunt zilele în care s-au emis mai multe facturi decât pe 2 august 2007?

```
SELECT DataFact AS Zi, COUNT(NrFact) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(NrFact) >
        (SELECT COUNT(NrFact)
        FROM facturi
        WHERE DataFact = DATE'2007-08-02')
```

Care este ziua în care s-au emis cele mai multe facturi?

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) >= ALL
        (SELECT COUNT(*)
        FROM facturi
        GROUP BY DataFact)
```

Subconsultarea calculează numărul de facturi corespunzător fiecărei zile. Predicatul clauzei HAVING compară numărul de facturi al fiecărei zile cu toate valorile extrase de subconsultare. Se obține tabela din figura 9.13.

DATAFACT	NR_FACTURILOR
01-08-2007	4
07-08-2007	4

Figura 9.13. Zilele în care s-au emis cele mai multe facturi

În PostgreSQL ne putem folosi fără rușine (doar nu este prima dată!) de clauza LIMIT:

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) =
        (SELECT COUNT(*)
        FROM facturi
        GROUP BY DataFact
        ORDER BY COUNT(*) DESC LIMIT 1)
```

Merită adăugat că numai Oracle permite și o soluție bazată pe incluziunea unei funcții în alta:

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) =
```

```
        (SELECT MAX(COUNT(*))
        FROM facturi
        GROUP BY DataFact)
```

iar în SQL Server se poate formula o soluţie elegantă bazată pe operatorul TOP:

```
SELECT TOP 1 DataFact, COUNT(*) AS Nr
FROM facturi
GROUP BY DataFact
ORDER BY Nr DESC
```

Surprinzător pentru unii (ca mine), soluţia funcţionează. ORDER BY se aplică grupurilor, iar ordonarea împreună cu opţiunea TOP extrag rezultatul corect, însă incomplet, întrucât chiar dacă sunt mai multe zile cu acelaşi număr maxim de facturi, interogarea furnizează doar unul. Aşa că varianta corectă şi completă este:

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) =
        (SELECT TOP 1 COUNT(*)
        FROM facturi
        GROUP BY DataFact
        ORDER BY COUNT(*) DESC)
```

Adăugăm şi soluţia DB2:

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) =
        (SELECT COUNT(*)
        FROM facturi
        GROUP BY DataFact
        ORDER BY COUNT(*) DESC
        FETCH FIRST 1 ROW ONLY)
```

Care este clientul care a cumpărat cele mai multe produse?

Analog problemei anterioare, formulăm o soluţie generală:

```
SELECT DenCl, COUNT(DISTINCT CodPr) AS "CateProduse"
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
GROUP BY DenCl
HAVING COUNT(DISTINCT CodPr) >= ALL
        (SELECT COUNT(DISTINCT CodPr)
        FROM facturi f INNER JOIN liniifact lf
```

```
            ON f.NrFact = lf.NrFact
    GROUP BY CodCl)
```

DENCL	CiteProduse
Client 3 SRL	4

Figura 9.14. Clientul care a cumpărat cele mai multe produse

una „Oracle only" (COUNT în MAX):

```
SELECT DenCl, COUNT(DISTINCT CodPr) AS "CateProduse"
FROM clienti c INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
GROUP BY DenCl
HAVING COUNT(DISTINCT CodPr) =
        (SELECT MAX(COUNT(DISTINCT CodPr))
        FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        GROUP BY CodCl)
```

o alta VFP/SQL Server (clauza TOP):

```
SELECT TOP 1 DenCl, COUNT(DISTINCT CodPr) AS CateProduse
FROM clienti c INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
GROUP BY DenCl
ORDER BY CateProduse DESC
```

una PostgreSQL (LIMIT):

```
SELECT DenCl, COUNT(DISTINCT CodPr) AS "CateProduse"
FROM clienti c INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact= lf.NrFact
GROUP BY DenCl
HAVING COUNT(DISTINCT CodPr) =
        (SELECT COUNT(DISTINCT CodPr)
        FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        GROUP BY CodCl
        ORDER BY COUNT(DISTINCT CodPr) DESC LIMIT 1)
```

iar în DB2 se înlocuieşte ultima linie cu:

```
            ...
        ORDER BY COUNT(DISTINCT CodPr) DESC FETCH FIRST 1 ROW ONLY)
```

Care este compartimentul cu cea mai bună medie a salariilor tarifare?

Se exclude din discuţie compartimentul DIRECŢIUNE în care apare, singur şi ferice, directorul general (sintaxa nu este acceptată de SQL Server din cauza funcţiei TRUNC):

```
SELECT Compart, TRUNC(AVG(SalTarifar),0) AS Medie_Sal
```

```
FROM personal2
WHERE Compart <> 'DIRECTIUNE'
GROUP BY Compart
HAVING AVG(SalTarifar) >= ALL
        (SELECT AVG(SalTarifar)
        FROM personal2
        WHERE Compart <> 'DIRECTIUNE'
        GROUP BY Compart)
```

COMPART	MEDIE_SAL
MARKETING	1383

Figura 9.15. Compartimentul cu cea mai bună medie a salariilor tarifare

Pentru aducerea aminte a structurilor alternative, putem folosi şi variantele:

```
SELECT Compart, TRUNC(AVG(
        CASE
        WHEN Compart <> 'DIRECTIUNE' THEN SalTarifar
        ELSE 0
        END),0) AS Medie_Sal
FROM personal2
GROUP BY Compart
HAVING AVG(CASE WHEN Compart <> 'DIRECTIUNE' THEN SalTarifar
                ELSE 0
                END) >= ALL
        (SELECT AVG(
        CASE
                WHEN Compart <> 'DIRECTIUNE'
                        THEN SalTarifar
                        ELSE 0
                END
        )
        FROM personal2
        GROUP BY Compart)
```

şi o variantă bazată pe DECODE-ul „oraclo-db2-ist":

```
SELECT Compart, TRUNC(AVG(
        DECODE(Compart, 'DIRECTIUNE', 0, SalTarifar)
                ),0) AS Medie_Sal
FROM personal2
GROUP BY Compart
HAVING AVG(DECODE(Compart, 'DIRECTIUNE', 0, SalTarifar))
        >= ALL
```

```
(SELECT AVG(DECODE (Compart, 'DIRECTIUNE', 0, SalTarifar))
FROM personal2
GROUP BY Compart)
```

Funcția TRUNC a fost necesară pentru afişarea numai a părții întregi din medie. De data aceasta (şi următoarele), trecem peste varianta Oracle, cu funcție inclusă în altă funcție, şi versiunile MS SQL Server.

Care este județul în care berea s-a vândut cel mai bine?

În tabela PRODUSE există un atribut ce reprezintă grupa în care se încadrează produsul respectiv. Berea este una dintre grupele îndrăgite.

```
SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Bere
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE Grupa = 'Bere'
GROUP BY Judet
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA))
        >= ALL
(SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Bere
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE Grupa = 'Bere'
GROUP BY Judet)
```

JUDET	VINZARI_BERE
Iasi	1896545,5

Figura 9.16. Județul cu cea mai bună vânzare a produselor din grupa Bere

Pe baza exemplelor anterioare se pot redacta soluțiile „proprietare" DB2, Oracle, PostgreSQL şi SQLServer.

Care sunt clienții cu valoarea vânzărilor peste medie?

Noțiunea de medie este destul de alunecoasă. În acest caz, media vânzărilor pe client se calculează împărțind vânzările totale la numărul clienților cărora li s-a trmis măcar o factură:

```
SELECT dencl AS client, TRUNC(SUM(cantitate * pretunit * (1 + proctva)),0) AS vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY dencl
HAVING SUM(cantitate * pretunit * (1 + proctva)) >= ALL
        (SELECT SUM(cantitate * pretunit * (1 + proctva))/COUNT(DISTINCT f.codcl)
        FROM  facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr)
ORDER BY 1
```

CLIENT	VINZARI
Client 1 SRL	15061538
Client 3 SRL	17413834
Client 4	9479109

Figura 9.17. Clienți cu vânzări peste medie

Extrageți factura cu valoarea imediat peste cea medie.

Privitor la tipul de medie care ne interesează, vom împărți valoarea totală a vânzărilor la numărul de facturi emise:

```
SELECT f.NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Valoare
FROM liniifact lf INNER JOIN produse p ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
GROUP BY f.NrFact
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) <= ALL
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                AS "Facturi_cu_val_peste_medie "
        FROM liniifact lf
                INNER JOIN produse p ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        GROUP BY f.NrFact
        HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA))          >
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                        COUNT(DISTINCT f.NrFact) AS Media
                FROM liniifact lf
                        INNER JOIN produse p ON p.CodPr = lf.CodPr
                        INNER JOIN facturi f ON lf.NrFact = f.NrFact
                )
```

```
) AND SUM(Cantitate * PretUnit * (1 + ProcTVA)) >
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/
        COUNT(DISTINCT f.NrFact) AS Media
        FROM liniifact lf
                INNER JOIN produse p ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        )
```

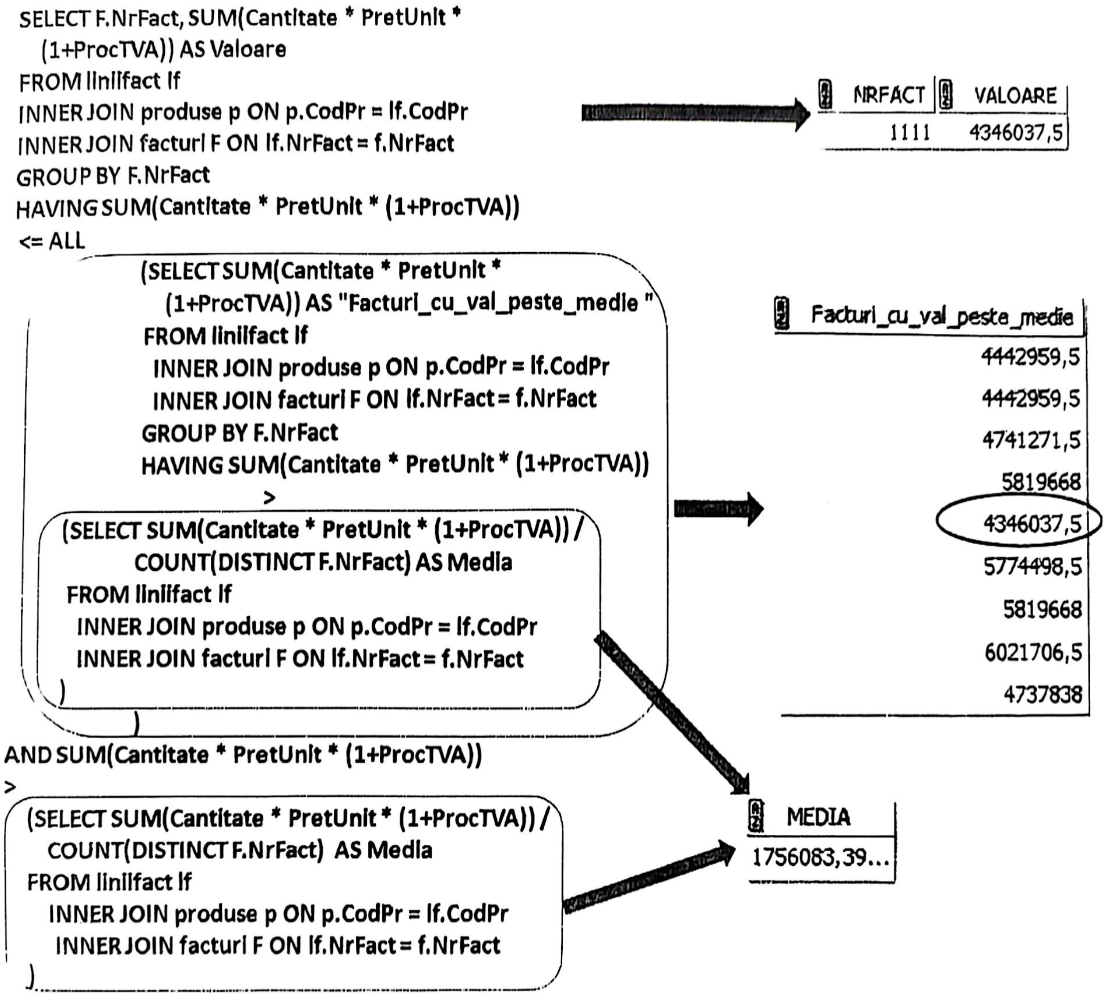


Figura 9.18. Factura cu cea mai mică valoare peste medie

Elementul de noutate al acestei interogări este includerea, într-o consultare ce prezintă clauza HAVING, a unei alte subconsultări în care apare, de asemenea, HAVING. Mecanismul execuției este explicat în figura 9.18, iar sintaxa este comună celor patru dialecte.

Care este județul cu vânzări imediat superioare județului Neamț?

```
SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
FROM judete j INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
```

```
          INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
          INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY Judet
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA))
          <= ALL
          (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
          FROM judete j INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                    INNER JOIN clienti c ON cp.CodPost = c.CodPost
                    INNER JOIN facturi f ON c.CodCl = f.CodCl
                    INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
          GROUP BY Judet
          HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA))
                    >
          (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
          FROM judete j
                    INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                    INNER JOIN clienti c ON cp.CodPost = c.CodPost
                    INNER JOIN facturi f ON c.CodCl=f.CodCl
                    INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
          WHERE Judet = 'Neamt'
          )
          )
          AND SUM(Cantitate * PretUnit * (1 + ProcTVA))
                    >
          (SELECT  SUM(Cantitate * PretUnit * (1 + ProcTVA))
          FROM judete j
                    INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                    INNER JOIN clienti c ON cp.CodPost = c.CodPost
                    INNER JOIN facturi f ON c.CodCl = f.CodCl
                    INNER JOIN liniifact lf ON f.NrFact= lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
          WHERE Judet = 'Neamt')
```

JUDET	VINZARI
Vaslui	17413834,5

Figura 9.19. Judeţul cu vânzări imediat superioare Neamţului

9.3.2. Exagerări (volumul n + 1)

Care este cel mai mare datornic dintre clienţi?

Este, cred, cea mai grea interogare de până acum. După cum am văzut atunci când am calculat pentru fiecare factură valorile facturate şi încasate, la joncţiunea externă a „părţii de facturare" cu „partea de încasare", fiecare tranşă de încasare se repetă în funcţie de numărul de linii ce compun factura respectivă, iar fiecare linie a facturii se repetă în funcţie de numărul de tranşe de încasare.

Trucul utilizat pentru a o scoate la capăt consta, pe de o parte, în împărţirea totalului tranşelor de încasare la numărul liniilor din factură, obţinând astfel valoarea încasată a facturii şi, pe de altă parte, împărţirea totalului valorilor liniilor din facturi la numărul tranşelor de încasare. Pentru problema de faţă trebuie găsită altă soluţie, deoarece gruparea nu o facem la nivel de factură, ci la nivel de client. Numărul de linii din facturi şi numărul de tranşe nu mai au nicio relevanţă la gruparea după client.

Pentru încasări, ar fi o idee. Deoarece o tranşă se repetă pentru fiecare linie a facturii, putem elimina din SUM toate tranşele pentru care *Linie* > *1*:

```
SELECT DenCl AS Client,
        SUM (
                CASE
                WHEN lf.Linie > 1 THEN NULL
                ELSE COALESCE(Transa,0)
                END
                ) AS Valoare_Incasata
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        LEFT OUTER JOIN incasfact i ON f.NrFact = i.NrFact
GROUP BY DenCl
```

Rezultatul din figura 9.20 ne încredinţează că ideea n-a fost chiar rea. Cu valoarea facturată, însă, lucrurile sunt mai complicate. Fiecare linie din factură se repetă de atâtea ori, în funcţie de câte tranşe de încasare există pentru factura respectivă. Cum gruparea se face pe clienţi, numărul tranşelor este irelevant. Trebuie însumate valorile distincte ale facturii şi liniei.

CLIENT	VALOARE_INCASATA
Client 1 SRL	286,490.00
Client 2 SA	106,275.00
Client 3 SRL	0.00
Client 4	0.00
Client 5 SRL	125,516.00
Client 6 SA	0.00
Client 7 SRL	0.00

Figura 9.20. Încasările pe clienţi

Este adevărat, clauza DISTINCT poate fi folosită şi cu SUM. Dar se poate întâmpla ca, la un moment dat, să existe într-o factură, două linii cu aceeaşi valoare. E musai de însumat valorile expresiei pentru combinaţii distincte *(NrFact, Linie)*. Confirmând vocaţia de popor de improvizatori (n-am zis cârpaci, să nu supăr adevăraţii patrioţi), recurgem la un truc ieftin, dar, vorba domnului Graur (comentatorul), năucitor!

```
SELECT DenCl,
        SUM( DISTINCT
                1000000000000000  * lf.NrFact +
                        1000000000000 * CAST (lf.Linie AS NUMERIC) +
        Cantitate * PretUnit * (1+ProcTVA))
AS Valoare_Facturata_Aiurea
FROM  produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
        LEFT OUTER JOIN  incasfact i ON f.NrFact=i.NrFact
GROUP BY DenCl
```

Nu putem discuta detalii înainte de a vedea ce a putut fi obţinut din interogare (vezi figura 9.21).

DENCL	VALOARE_FACTURATA_AIUREA
Client 1 SRL	51,874,040,000,015,061,538.0000
Client 2 SA	6,339,003,000,000,361,335.0000
Client 3 SRL	25,428,030,000,017,413,834.5000
Client 4	6,484,006,000,009,479,109.5000
Client 5 SRL	12,672,009,000,000,432,400.0000
Client 6 SA	3,342,006,000,006,021,706.5000
Client 7 SRL	6,348,003,000,000,400,411.5000

Figura 9.21. Improvizaţie pentru calculul valorii vânzărilor pe clienţi

Expresia de calcul a valorii facturate introduce, pe lângă Cantitate, PreţUnit şi ProcTVA, şi NrFact, şi Linie. Ultimele două atribute sunt înmulţite cu constante foarte mari (10^{15}, respectiv 10^{12}), suficient de mari pentru ca valoarea propriu-zisă a facturilor să nu fie „afectată". Este sigur că valorile obţinute pentru fiecare client au fost calculate luându-se în calcul o singură dată fiecare linie dintr-o factură. Iar dacă privim în rezultat partea din dreapta fiecărui număr (după cele patru sau mai multe zerouri), observăm că acelea sunt chiar valorile facturate care ne interesează. Prin urmare, nu ne mai rămâne decât să extragem acea parte:

```
SELECT DenCl,
        SUM(DISTINCT
                1000000000000000 * lf.NrFact +
                1000000000000 * CAST (lf.Linie AS NUMERIC) +
                Cantitate * PretUnit * (1+ProcTVA)) AS Valoare_Umflata,
        CAST(
                RIGHT(CAST(SUM(DISTINCT
                1000000000000000 * lf.NrFact +
                1000000000000 * CAST (lf.Linie AS NUMERIC) +
                Cantitate * PretUnit * (1 + ProcTVA)) AS CHAR(35)), 19)
                        AS DECIMAL (16,4)) AS Valoare_Fact_Corect
FROM produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
        LEFT OUTER JOIN  incasfact i ON f.NrFact = i.NrFact
GROUP BY DenCl
```

Obținem ceea ce doream, adică valoarea vânzărilor pe clienți, după cum o arată figura 9.22.

DENCL	VALOARE_UMFLATA	VALOARE_FACT_CORECT
Client 1 SRL	51,874,040,000,015,061,538.0000	15,061,538.0000
Client 2 SA	6,339,003,000,000,361,335.0000	361,335.0000
Client 3 SRL	25,428,030,000,017,413,834.5000	17,413,834.5000
Client 4	6,484,006,000,009,479,109.5000	9,479,109.5000
Client 5 SRL	12,672,009,000,000,432,400.0000	432,400.0000
Client 6 SA	3,342,006,000,006,021,706.5000	6,021,706.5000
Client 7 SRL	6,348,003,000,000,400,411.5000	400,411.5000

Figura 9.22. Valoarea vânzărilor, pe clienți

În final, soluția problemei (şi răspunsul în figura 9.23):

```
SELECT DenCl,
        CAST(
                RIGHT(CAST(SUM(DISTINCT 1000000000000000 * lf.NrFact +
                                1000000000000 * CAST (lf.Linie AS NUMERIC) +
                                Cantitate * PretUnit * (1 + ProcTVA)
                                ) AS CHAR(35))
                        , 19)
                AS DECIMAL (16,4)) AS Valoare_Facturata,
        SUM(CASE WHEN lf.Linie > 1 THEN NULL
                                ELSE COALESCE(Transa,0) END
                ) AS Valoare_Incasata,
```

```
        CAST(
                RIGHT(CAST(SUM(DISTINCT 1000000000000000 * lf.NrFact
                        + 1000000000000 * CAST (lf.Linie AS NUMERIC) +
                        Cantitate * PretUnit * (1 + ProcTVA))
                                AS CHAR(35))
                        , 19)
                AS DECIMAL (16,4))
                -
                SUM(CASE WHEN lf.Linie > 1 THEN NULL
                        ELSE COALESCE(Transa,0) END )
        AS Rest_De_Incasat
FROM produse P
        INNER JOIN liniifact lf ON P.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
        LEFT OUTER JOIN incasfact i ON f.NrFact = i.NrFact
GROUP BY DenCl
HAVING
        (CAST(
                RIGHT(CAST(SUM(DISTINCT
                        1000000000000000 * lf.NrFact + 1000000000000 *
                                CAST (lf.Linie AS NUMERIC) + Cantitate *
                                PretUnit * (1 + ProcTVA))
                                AS CHAR(35))
                        , 19)
                AS DECIMAL (16,4))
                -
        SUM(CASE WHEN lf.Linie > 1 THEN NULL ELSE COALESCE(Transa,0) END))
                >= ALL
        (SELECT CAST(RIGHT(CAST(SUM(DISTINCT
                1000000000000000 * lf.NrFact + 1000000000000 *
                CAST(lf.Linie AS NUMERIC) +
                Cantitate * PretUnit * (1 + ProcTVA))
                                        AS CHAR(35))
                                , 19)
                        AS DECIMAL (16,4))
        -
        SUM(CASE WHEN lf.Linie > 1 THEN NULL ELSE COALESCE(Transa,0) END
                )
        FROM produse p
```

```
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
        LEFT OUTER JOIN  incasfact i ON f.NrFact = i.NrFact
    GROUP BY DenCl)
```

DENCL	VALOARE_FACTURATA	VALOARE_INCASATA	REST_DE_INCASAT
Client 3 SRL	17,413,834.5000	0.00	17,413,834.5000

Figura 9.23. Clientul ce are cel mai mare rest de plată

După atâta strădanie, este destul de greu de suportat vestea că soluția funcționează doar în DB2. Astfel, SQL Server nu face nazuri la execuție, dar afişează ce ştie el (vezi figura 9.24). Necazul vine de la modul de extragere şi conversie a şirului de caractere obținut prin expresia-artificiu pentru calcularea valorii facturate. Dacă înlocuim toate argumentele funcțiilor RIGHT din *RIGHT ..., 19)* în *RIGHT(..., 27)*, rezultatul este cel corect.

DenCl	Valoare_Facturata	Valoare_Incasata	Rest_De_Incasat
Client 3 SRL	3834.5000	0.00	3834.50

Figura 9.24. Rezultatul interogării de mai sus executată în SQL Server

Nici Oracle şi nici PostgreSQL nu au implementată funcția RIGHT; de aceea, este necesară înlocuirea acesteia cu SUBSTR (plus, pentru siguranță, funcția LPAD). Dar am exagerat destul. Revenim în paragraful 9.4 cu o soluție rezonabilă pentru această problemă.

9.3.3. Grupuri şi diviziuni relaționale

Revenind la cazul teoretic din paragraful 4.4.7 (figura 4.30), câtul diviziunii relaționale a RD1 : RD2 (altfel spus, găsirea tuturor icşilor care sunt în RD1 în combinație cu toți igrecii din RD2), se calculează în SQL şi astfel:

```
SELECT X
FROM rd1
GROUP BY X
HAVING COUNT(Y) =
        (SELECT COUNT(Y)
        FROM rd2)
```

Pentru ce clienți există cel puțin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007? (exemplul 23, paragraf 4.4.7)?

```
SELECT DenCl, COUNT(DISTINCT DataFact)
FROM facturi f INNER JOIN clienti c ON f.CodCl = c.CodCl
WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
```

```
GROUP BY DenCl
HAVING COUNT(DISTINCT DataFact) =
        (SELECT COUNT (DISTINCT DataFact)
        FROM facturi
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        )
```

Celelalte interogări din acest paragraf respectă sintaxa tuturor celor patru dialecte. În schimb, aceasta necesită eliminarea cuvintelor DATE pentru a fi executabilă în DB2 şi SQL Server.

Ce produse au fost vândute tuturor clienţilor? (exemplul 26, paragraful 4.4.7)

```
SELECT DenPr
FROM produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
GROUP BY DenPr
HAVING COUNT(DISTINCT CodCl) =
        (SELECT COUNT (CodCl)
        FROM clienti)
```

Care sunt facturile ce conţin măcar produsele din factura 1117? (exemplul 27, paragraful 4.4.7)

Problema pare mai dificilă decât precedentele, pentru că trebuie numărate, din fiecare factură, numai produsele care se găsesc în factura-etalon 1117:

```
SELECT lf1.NrFact
FROM liniifact lf1
        INNER JOIN liniifact lf2 ON lf1.CodPr = lf2.CodPr AND lf2.NrFact = 1117
GROUP BY lf1.NrFact
HAVING COUNT(DISTINCT lf1.CodPr) =
        (SELECT COUNT(DISTINCT CodPr)
        FROM liniifact
        WHERE NrFact = 1117)
```

9.4. Subconsultări în clauza FROM

Posibilitatea de a defini tabele „ad-hoc", în clauza FROM, este unul dintre cele mai importante daruri pe care SQL-ul îl oferă utilizatorilor, deopotrivă profesionişti şi neprofesionişti în ale bazelor de date.

9.4.1. Subconsultări pentru filtrare, calcule şi grupări

Ce facturi au fost emise în aceeaşi zi cu factura 1120?

Nu este cel mai de efect mod de a introduce în scenă subconsultările din clauza FROM, dar ce ziceți de soluția următoare?

```
SELECT f.*
FROM facturi f INNER JOIN
        (SELECT *
        FROM facturi
        WHERE NrFact = 1120
        ) f1120
ON f.DataFact = f1120.DataFact
```

Dacă până acum în clauza FROM joncționam numai tabele ale bazei, în această interogare joncționăm (după atributul DataFact) o tabelă (FACTURI) cu o tabelă temporară (F1120), definită ad-hoc printr-o subconsultare. După cum o arată şi figura 9.25, tabela F1120 are o singură linie, cea corespunzătoare facturii-etalon (1120), de aici şi numele său.

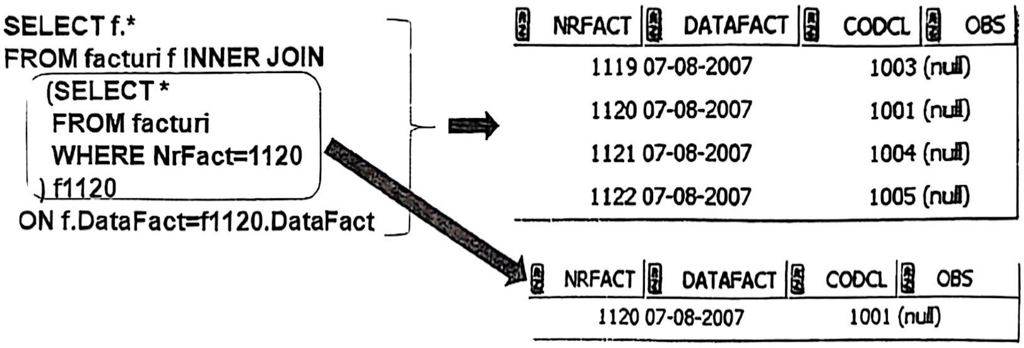


Figura 9.25. Joncțiunea unei tabele cu rezultatul unei subconsultări

În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"?

În clauza FROM vom defini două subconsultări, una care va extrage zilele în care s-a vândut „Produs 1" (ZILE_PROD1) şi cealaltă destinată zilelor în care s-a vândut „Produs 2" (ZILE_PROD2). Rezultatele celor două subconsultări vor fi joncționate după atributul DataFact:

```
SELECT DISTINCT zile_prod1.DataFact
FROM
(SELECT DataFact
FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE DenPr = 'Produs 1') zile_prod1
        INNER JOIN
(SELECT DataFact
```

```
FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE DenPr = 'Produs 2') zile_prod2
        ON zile_prod1.DataFact = zile_prod2.DataFact
ORDER BY 1
```

Care dintre angajaţi au salariul tarifar egal cu cel al ANGAJAT2?

Lucrăm după acelaşi calapod:

```
SELECT NumePren FROM
        (SELECT NumePren, SalTarifar
        FROM personal2) temp1
        INNER JOIN
                    (SELECT SalTarifar
                    FROM personal2
                    WHERE NumePren = 'ANGAJAT 2') temp2
              ON temp1.SalTarifar = temp2.SalTarifar
```

Adevărata forţă a acestei opţiuni se observă însă în interogările complexe, în care liniile tabelelor temporare reprezintă nu tupluri, ci grupuri de tupluri. Practic, în loc să se compare două subconsultări în clauza HAVING (clauză, care, la rândul său, presupune GROUP BY), includem gruparea în subconsultările din FROM, iar comparaţia o facem între tuplurile tabelelor temporare ad-hoc.

Care sunt valorile facturate şi încasate, precum şi situaţia („fără nici o încasare", „încasată parţial" sau „încasată total") pentru fiecare factură?

```
SELECT VANZARI.NrFact, Facturat, COALESCE(Incasat,0) AS Incasat,
        Facturat - COALESCE(Incasat,0) AS Diferenta,
        CASE
              WHEN  COALESCE(Incasat,0) = 0 THEN 'Fara nici o incasare'
              WHEN Facturat > COALESCE(Incasat,0) THEN 'Incasata partial'
              ELSE 'INCASATA TOTAL'
        END AS Situatiune
FROM  (
        SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat
        FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY NrFact
        ) VANZARI
   LEFT OUTER JOIN
        (
        SELECT NrFact, SUM(Transa) AS Incasat
        FROM incasfact
```

```
    GROUP BY NrFact
    ) INCASARI
  ON VANZARI.NrFact = INCASARI.NrFact
```

În clauza FROM a frazei SELECT principale au fost definite două tabele temporare, VÂNZĂRI şi ÎNCASĂRI. Prima conţine valoarea totală a fiecărei facturi, în timp ce a doua valoarea încasată. Aceste două tabele sunt joncţionate extern (pentru a include şi facturile fără nicio încasare) după atributul NrFact (vezi figura 9.26).

```
SELECT VINZARI.Nrfact, Facturat,
   COALESCE(Incasat,0) AS Incasat,
   Facturat - COALESCE(Incasat,0)
     AS Diferenta,
 CASE
  WHEN  COALESCE(Incasat,0) = 0
    THEN 'Fara nici o incasare'
  WHEN Facturat > COALESCE(Incasat,0)
    THEN 'Incasata partial'
  ELSE 'INCASATA TOTAL'
 END AS Situatiune
FROM
 (
  SELECT NrFact, SUM(
    Cantitate * PretUnit * (1+
    ProcTVA)) AS Facturat
  FROM liniifact lf
    INNER JOIN produse p
      ON lf.CodPr = p.CodPr
  GROUP BY NrFact
 ) VINZARI
  LEFT OUTER JOIN
 (
  SELECT NrFact,
    SUM(Transa) AS Incasat
  FROM incasfact
  GROUP BY NrFact
  ) INCASARI
  ON VINZARI.NrFact
    = INCASARI.NrFact
```

NRFACT	FACTURAT
1118	201975
3111	4442959,5
3115	110907,5
2112	153442
3118	179565

...continuare...

NRFACT	INCASAT
1118	201975
1112	125516
1113	106275
1111	53996
1120	7315
1117	23204

VINZARI.NrFact	FACTURAT	INCASARI.NrFact	INCASAT
1118	201975	1118	201975
1112	125516	1112	125516
1113	106275	1113	106275
1111	4346037,5	1111	53996
1120	97664	1120	7315
1117	222050	1117	23204
3117	287655	(null)	(null)
2113	127530	(null)	(null)
1114	6021706,5	(null)	(null)
3112	153442	(null)	(null)

...continuare...

NRFACT	FACTURAT	INCASAT	DIFERENTA	SITUATIUNE
1118	201975	201975	0	INCASATA TOTAL
1112	125516	125516	0	INCASATA TOTAL
1113	106275	106275	0	INCASATA TOTAL
1111	4346037,5	53996	4292041,5	Incasata partial
1120	97664	7315	90349	Incasata partial
1117	222050	23204	198846	Incasata partial
3117	287855	0	287855	Fara nici o incasare
2113	127530	0	127530	Fara nici o incasare
1114	6021706,5	0	6021706,5	Fara nici o incasare
3112	153442	0	153442	Fara nici o incasare

...continuare...

Figura 9.26. Subconsultări în clauza FROM

Să se obţină sporurile de noapte pentru al doilea trimestru al anului 2007, atât lunar, cât şi cumulat.

```
SELECT sl1.Marca, NumePren,
        COALESCE(sl1.SpN,0) AS Sp_N_Apr,
        COALESCE(sl2.SpN,0) AS Spor_N_Mai,
        COALESCE(sl3.SpN,0) AS Spor_N_Iun ,
        COALESCE(sl1.SpN,0) + COALESCE(sl2.SpN,0) + COALESCE(sl3.SpN,0)
            AS Sp_Noapte_Trim2
FROM
        (SELECT p.Marca, NumePren, SporNoapte AS SpN
```

```
FROM personal2 p LEFT OUTER JOIN sporuri s
        ON p.Marca = s.Marca AND An = 2007  AND Luna = 4
) sl1 INNER JOIN
(SELECT p.Marca, SporNoapte AS SpN
FROM personal2 p LEFT OUTER JOIN sporuri s ON p.Marca = s.Marca
        AND An = 2007 AND Luna = 5
) sl2 ON sl1.Marca = sl2.Marca
        INNER JOIN
        (SELECT p.Marca, SporNoapte AS SpN
        FROM personal2 p
        LEFT OUTER JOIN sporuri s ON p.Marca = s.Marca
        AND An = 2007 AND Luna = 6
) sl3 ON sl1.Marca = sl3.Marca
ORDER BY NumePren
```

Clauza FROM principală „calculează" trei tabele temporare ce conțin sporurile de noapte pe lunile aprilie (SL1), mai (SL2) şi iunie (SL3) 2005 ale fiecărui angajat (indiferent de data angajării acestuia). Cele trei tabele sunt joncționate după atributul Marca (vezi figura 9.27).

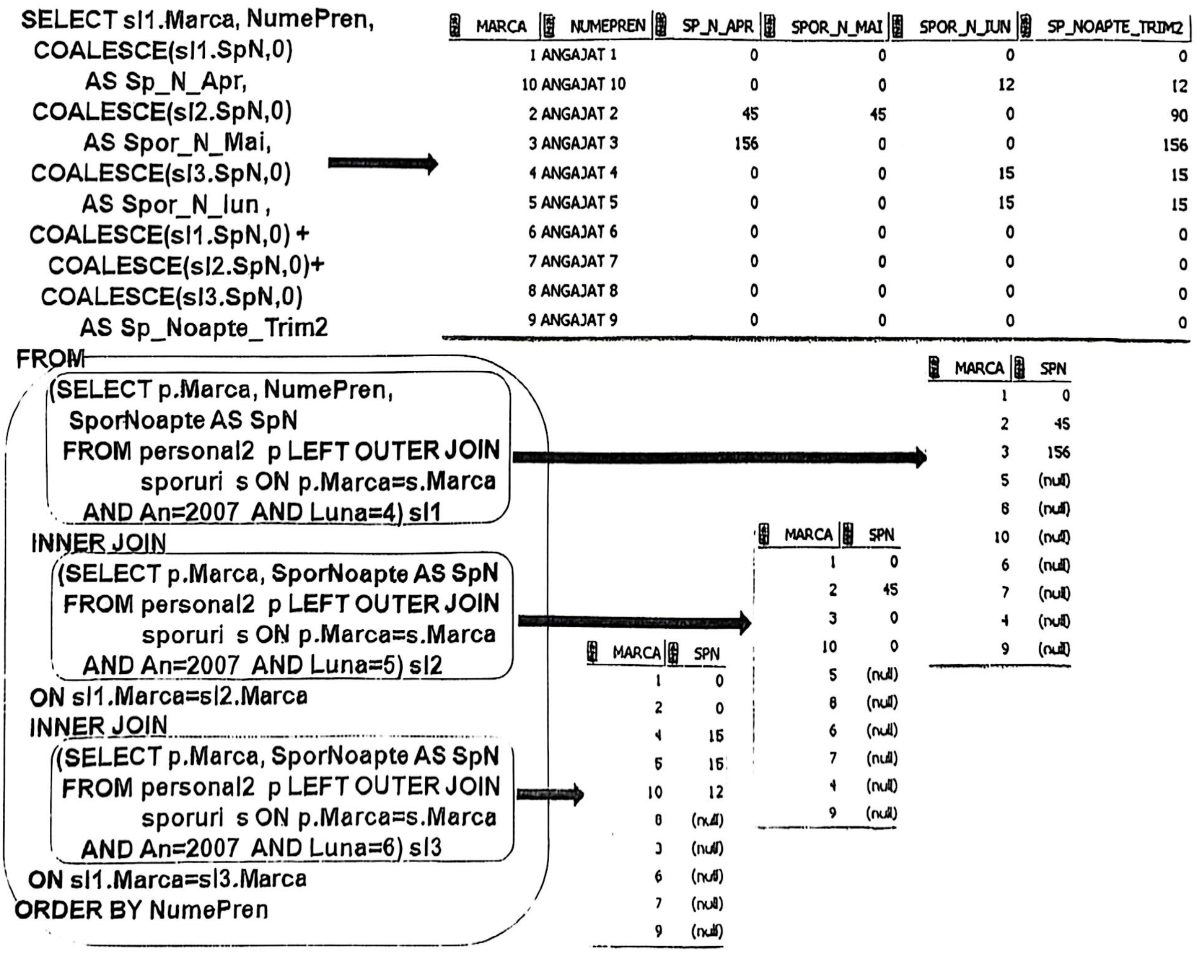


Figura 9.27. Sporuri de noapte pe trimestrul II în 2007

Să se afişeze câte facturi sunt: neîncasate deloc, încasate parţial şi încasate total?

```
SELECT
        CASE
                WHEN COALESCE(Incasat,0) = 0 THEN 'Fara nici o incasare'
                WHEN Facturat > COALESCE(Incasat,0) THEN 'Incasata partial'
                ELSE 'INCASATA TOTAL'
        END AS Situatiune, COUNT(*) AS Nr
FROM (SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat
        FROM liniifact LF INNER JOIN produse P ON LF.CodPr = P.CodPr
        GROUP BY NrFact
        ) VANZARI
    LEFT OUTER JOIN
        (SELECT NrFact, SUM(Transa) AS Incasat
        FROM incasfact
        GROUP BY NrFact
        ) INCASARI
    ON VANZARI.NrFact = INCASARI.NrFact
GROUP BY
        CASE
                WHEN COALESCE(Incasat,0) = 0 THEN 'Fara nici o incasare'
                WHEN Facturat > COALESCE(Incasat,0) THEN 'Incasata partial'
                ELSE 'INCASATA TOTAL'
        END
```

Joncţiunea externă la stânga dintre VÂNZĂRI şi ÎNCASĂRI este completată de o structură CASE.

Să se determine discountul financiar pentru fiecare tranşă de factură încasată, ştiind că acesta se acordă pentru clienţii care plătesc rapid facturile şi se calculează pe tranşe, astfel:

- pentru tranşele încasate în maximum 9 zile de la data facturării, procentul este de 10%;
- pentru tranşele încasate în intervalul 10-12 zile de la data facturării, procentul este de 7%;
- pentru tranşele încasate în intervalul 13-15 zile de la data facturării, procentul este de 5%;
- pentru tranşele încasate în intervalul 16-18 zile de la data facturării, procentul este de 3%;
- pentru tranşele încasate după 19 zile nu se acordă niciun bonus.

Trebuie să recunosc că găsesc problema tentantă şi pentru că pot exemplifica includerea, pe mai multe nivele, a subconsultărilor în clauza FROM. În plus, pentru aceeaşi factură pot exista oricâte tranşe de încasare, fiecare tranşă fiind subiect de discount financiar în funcţie de momentul încasării. Începem cu o interogarea pregătitoare:

```
SELECT NrFact, DataFact, Facturat, DataInc, Incasat, NrZile, Procent_DscntFin,
        Incasat * Procent_DscntFin AS Discount_Fin
FROM
        (SELECT vanzari.NrFact, DataFact, Facturat,
                COALESCE(DataInc, CURRENT_DATE) AS DataInc
                ,COALESCE(Incasat,0) AS Incasat,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        AS NrZile,
                CASE
                WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        < 10 THEN 0.10
                WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        < 13 THEN 0.07
                WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        < 16 THEN 0.05
                WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        < 19 THEN 0.03
                ELSE 0
                END AS Procent_DscntFin
        FROM
        (
        SELECT f.NrFact, DataFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Facturat
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY f.NrFact, DataFact
        ) VANZARI
                LEFT OUTER JOIN
        (
        SELECT NrFact, DataInc, ROUND(SUM(Transa),0) AS Incasat
        FROM  incasfact INNER JOIN incasari ON incasfact.CodInc = incasari.CodInc
        GROUP BY NrFact, DataInc
        ) INCASARI
                ON VANZARI.NrFact = INCASARI.NrFact
        ORDER BY NrFact, DataInc
) x
```
[4]

4. Sintaxa nu este acceptată de SQL Server din cauza funcţiilor CURRENT_DATE şi TRUNC.

Rezultatul este promițător (primele linii din rezultat se găsesc în figura 9.28). Atât tabela ad-hoc VÂNZĂRI, cât și ÎNCASĂRI prezintă atributele dedicate datei facturării și datei încasării tranșei respective. Diferența dintre cele două date reprezintă numărul de zile în funcție de care se acordă discountul.

NRFACT	DATAFACT	FACTURAT	DATAINC	INCASAT	NRZILE	PROCENT_DSCNTFIN	DISCOUNT_FIN
1111	01-08-2007	4346038	15-08-2007	53996	14	0,05	2699,8
1112	01-08-2007	125516	15-08-2007	125516	14	0,05	6275,8
1113	01-08-2007	106275	17-08-2007	106275	16	0,03	3188,25
1114	01-08-2007	6021707	12-03-2008	0	224	0	0
1115	02-08-2007	151238	12-03-2008	0	223	0	0
1116	02-08-2007	126713	12-03-2008	0	223	0	0
1117	03-08-2007	222050	16-08-2007	9754	13	0,05	487,7
1117	03-08-2007	222050	17-08-2007	9754	14	0,05	487,7
1117	03-08-2007	222050	18-08-2007	3696	15	0,05	184,8
1118	04-08-2007	201975	15-08-2007	101975	11	0,07	7138,25
1118	04-08-2007	201975	16-08-2007	100000	12	0,07	7000
1119	07-08-2007	5774499	12-03-2008	0	218	0	0
1120	07-08-2007	97664	16-08-2007	7315	9	0,1	731,5
1121	07-08-2007	4737838	12-03-2008	0	218	0	0

Figura 9.28. Calcularea discountului financiar pe fiecare tranșă de încasare

Răspunsul exact la problema formulată presupune gruparea după numărul facturii și însumarea atât a tranșelor încasate, cât și a discounturilor financiare pentru fiecare tranșă. Atenție, însă, nu trebuie însumată și valoarea totală a facturii, așa încât interogarea va avea forma:

```
SELECT NrFact, DataFact, Facturat, SUM(Incasat) AS Total_Incasat,
        SUM(Discount_Fin) AS Total_Discount_Fin
FROM
        (
        SELECT NrFact, DataFact, Facturat, DataInc, Incasat,
                NrZile, Procent_DscntFin,
                ROUND(Incasat * Procent_DscntFin,0) AS Discount_Fin
        FROM
                (SELECT vanzari.NrFact, DataFact, Facturat,
                        COALESCE(DataInc, CURRENT_DATE) AS DataInc
                        ,COALESCE(Incasat,0) AS Incasat,
                        TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                                AS NrZile,
                        CASE
                        WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) –
                                DataFact,0) <= 9 THEN 0.10
                        WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) –
                                DataFact,0) <= 12 THEN 0.07
```

```
                    WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) -
                              DataFact,0) <= 15 THEN 0.05
                    WHEN TRUNC(COALESCE(DataInc, CURRENT_DATE) -
                              DataFact,0) <= 18 THEN 0.03
                    ELSE 0
                    END AS Procent_DscntFin
          FROM
                    (
                    SELECT f.NrFact, DataFact, ROUND(SUM(
                              Cantitate * PretUnit * (1 + ProcTVA)),0) AS Facturat
                    FROM facturi f
                              INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                              INNER JOIN produse p ON lf.CodPr = p.CodPr
                    GROUP BY f.NrFact, DataFact
                    ) VANZARI
                                        LEFT OUTER JOIN
                    (
                    SELECT NrFact, DataInc, ROUND(SUM(Transa),0) AS Incasat
                    FROM incasfact INNER JOIN incasari
                              ON incasfact.CodInc = incasari.CodInc
                    GROUP BY NrFact, DataInc
                    ) INCASARI
                                        ON VANZARI.NrFact = INCASARI.NrFact
          ORDER BY NrFact, DataInc
                    ) X
     ) Y
GROUP BY NrFact, DataFact, Facturat
ORDER BY 1
```

Uşoarele diferenţe de la însumare (vezi figura 9.29 comparativ cu figura 9.28) se datorează rotunjirilor (funcţia ROUND). Probabil că cea mai importantă factură din rezultat este 1117 care are trei tranşe de încasare, fiecare cu discountul său.

NRFACT	DATAFACT	FACTURAT	TOTAL_INCASAT	TOTAL_DISCOUNT_FIN
1111	01-08-2007	4346038	53996	2700
1112	01-08-2007	125516	125516	6276
1113	01-08-2007	106275	106275	3188
1114	01-08-2007	6021707	0	0
1115	02-08-2007	151238	0	0
1116	02-08-2007	126713	0	0
1117	03-08-2007	222050	23204	1161
1118	04-08-2007	201975	201975	14138
1119	07-08-2007	5774499	0	0
1120	07-08-2007	97664	7315	732
1121	07-08-2007	4737838	0	0

Figura 9.29. Totalizarea discountului financiar pe fiecare factură

Cu ocazia aceasta descoperim că, dacă eram aşa de generoşi, ar fi trebuit să restituim nişte bani unor clienţi.

Să se calculeze penalizările pentru întârzierea la plată a facturilor, ştiind că şi acestea se calculează pentru fiecare zi de întârziere, pe tranşe, astfel:

- pentru tranşele neîncasate în maximum 10 zile de la data facturării, nu se calculează penalizări;
- pentru tranşele neîncasate în intervalul 11-12 zile de la data facturării, procentul de penalitate este 0,05% din valoarea neîncasată, pentru fiecare zi de întârziere;
- pentru tranşele neîncasate în intervalul 13-14 zile de la data facturării, procentul de penalitate este 0,1% din valoarea neîncasată, pentru fiecare zi de întârziere;
- pentru tranşele încasate în intervalul 15-20 de zile de la data facturării, procentul este de 0,2% din valoarea neîncasată pentru fiecare zi de întârziere;
- pentru tranşele încasate în intervalul 20-40 de zile de la data facturării, procentul este de 0,3% din valoarea neîncasată pentru fiecare zi de întârziere;
- pentru tranşele încasate cu întârziere de peste 40 de zile de la data facturării, procentul este de 0,5% din valoarea neîncasată pentru fiecare zi de întârziere.

După cum vedeţi, acum am trecut în extrema cealaltă, a cărpănoşilor. Necazul este că, pentru această problemă nu suntem în stare să formulăm un răspuns corect, întrucât o factură poate avea mai multe tranşe de încasare, iar penalitatea depinde de restul de încasat din momentul fiecărei tranşe, rest calculat prin diferenţa dintre valoarea totală a facturii şi valoarea cumulată a tranşelor de încasare pentru factura respectivă până în momentul respectiv. Vom avea ceva mai mult succes în capitolul următor (capitolul 10) şi în capitolul dedicat opţiunilor OLAP (capitolul 11).

9.4.2. Un artificiu „proprietar" - ROWNUM din Oracle

Cu atâtea kilograme de SQL înghiţite, putem să realizăm că interogarea:

```
SELECT DISTINCT PretUnit
FROM liniifact
```

```
ORDER BY PretUnit DESC
```

extrage în ordine descrescătoare toate valorile (distincte) ale preţului unitar la care au fost vândute produsele. Adăugăm, discret, o clauză proprietară Oracle, ROWNUM, care nu face mare lucru - atribuie un număr fiecărei linii din rezultat:

```
SELECT DISTINCT PretUnit, ROWNUM
FROM liniifact
ORDER BY PretUnit DESC
```

Din figura 9.30 aflăm o veste bună - fiecare linie prezintă un număr - şi o veste rea - numărul respectiv nu ne foloseşte la nimic.

PRETUNIT	ROWNUM
7064	9
7064	21
7064	39
7060	3
7060	25
7060	43
6300	19
6300	38
6300	56
1705	8
1410	18
1410	37

Figura 9.30. Clauza ROWNUM (aplicată greşit)

Apelând la o subconsultare în clauza FROM, lucrurile devin ceva mai interesante:

```
SELECT t.*, ROWNUM
FROM
        (SELECT DISTINCT PretUnit
        FROM liniifact
        ORDER BY PretUnit DESC
        ) t
```

Tabela ad-hoc T, obţinută prin subconsultare, conţine valorile distincte ale preţului unitar, ordonate descrescător. Acum, ROWNUM va aloca valoarea 1 pentru cel mai mare preţ unitar, valoarea 2 pentru următorul ş.a.m.d (vezi figura 9.31).

PRETUNIT	ROWNUM
7064	1
7060	2
6300	3
1705	4
1410	5
1120	6
1100	7
1090	8
1070	9
1050	10
1030	11
1000	12
975	13
950	14
930	15
925	16
750	17
700	18

Figura 9.31. Clauza ROWNUM (aplicată ceva mai bine)

În continuare, dacă dorim să aflăm cele mai mari cinci prețuri unitare din LINIIFACT, aplicăm condiția de filtrare asupra coloanei ROWNUM:

```
SELECT t.*, ROWNUM
FROM
        (SELECT DISTINCT PretUnit
        FROM liniifact
        ORDER BY PretUnit DESC) t
WHERE ROWNUM <= 5
```

Atenție, însă, mânați de dorință, am fi tentați să credem că al cincilea (în ordine descrescătoare) preț unitar s-ar putea afla eliminând din interogarea precedentă semnul <:

```
SELECT t.*, ROWNUM
FROM
        (SELECT DISTINCT PretUnit
        FROM liniifact
        ORDER BY PretUnit DESC) t
WHERE ROWNUM = 5
```

Rezultatul nu are nici o linie, întrucât atribuirea numărului de linie se face *după* filtrare. Necazul se păstrează şi când includem interogarea de mai sus în clauza FROM a unei consultări:

```
SELECT *
FROM
```

```
        (SELECT t.*, ROWNUM
        FROM
                (SELECT DISTINCT PretUnit
                FROM liniifact
                ORDER BY PretUnit DESC) t
        )
WHERE ROWNUM = 5
```

Ne amăgim în van, întrucât ROWNUM-ul din clauza WHERE se aplică la nivelul noii interogări (nu e ROWNUM-ul pentru T). Din fericire, reparaţia este minoră: în tabela ad-hoc T redenumim coloana *ROWNUM* în *Nr* şi aplicăm condiţia de filtrare la noul atribut „calculat":

```
SELECT *
FROM (SELECT t.*, ROWNUM AS Nr
        FROM
                (SELECT DISTINCT PretUnit
                FROM liniifact
                ORDER BY PretUnit DESC) t
        )
WHERE Nr = 5
```

Care sunt cele mai mari cinci preţuri unitare de vânzare, produsele şi facturile în care apar cele cinci preţuri maxime?

Această problemă la a cărei premieră am avut ceva frisoane (paragraful 9.2) se rezolvă „cu vorba bună" astfel:

```
SELECT NrFact, DenPr, PretUnit
FROM liniifact INNER JOIN produse ON liniifact.CodPr = produse.CodPr
WHERE PretUnit IN
        (SELECT PretUnit
        FROM
                (SELECT t.*, ROWNUM
                FROM
                        (SELECT DISTINCT PretUnit
                        FROM liniifact
                        ORDER BY PretUnit DESC
                        ) t
                WHERE ROWNUM <=5
                )
```

9.4.3. Comparații mutate din HAVING în clauza FROM

Fireşte, ne putem gândi de acum cu mai multă simpatie şi la problemele ce necesitau comparații în clauza HAVING.

Care sunt zilele în care s-au emis mai multe facturi decât pe 2 august 2007?

```
SELECT Zi, Nr_Facturilor
FROM
        (SELECT DataFact AS Zi, COUNT(NrFact) AS Nr_Facturilor
        FROM facturi
        GROUP BY DataFact ) fact_zile
                INNER JOIN
                (SELECT COUNT(NrFact) AS NrFact_2Aug
                FROM facturi
                WHERE DataFact = DATE'2007-08-02') fact_2aug
        ON Nr_Facturilor > NrFact_2Aug
```

Care este ziua în care s-au emis cele mai multe facturi?

Din păcate (sau, din fericire!), soluția:

```
SELECT temp1.*
FROM
        (SELECT DataFact, COUNT(Nrfact) AS Nr
        FROM facturi
        GROUP BY DataFact) temp1,
                (SELECT DataFact, COUNT(Nrfact) AS Nr
                 FROM facturi
                 GROUP BY DataFact) temp2
WHERE temp1.Nr >= ALL           (SELECT Nr FROM temp2)
```

nu funcționează (vezi figura 9.32). Aceasta înseamnă că o tabelă temporară definită pe baza unei subconsultări în clauza FROM nu este recunoscută într-o sub-consultare din clauza WHERE sau HAVING. Se poate, totuşi, utiliza cu succes varianta:

```
SELECT DataFact, COUNT(*) AS Nr_Facturi
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) = (SELECT MAX(Nr)
                FROM (SELECT DataFact, COUNT(NrFact) AS Nr
                        FROM facturi
                        GROUP BY DataFact
                        ) TEMP1)
```

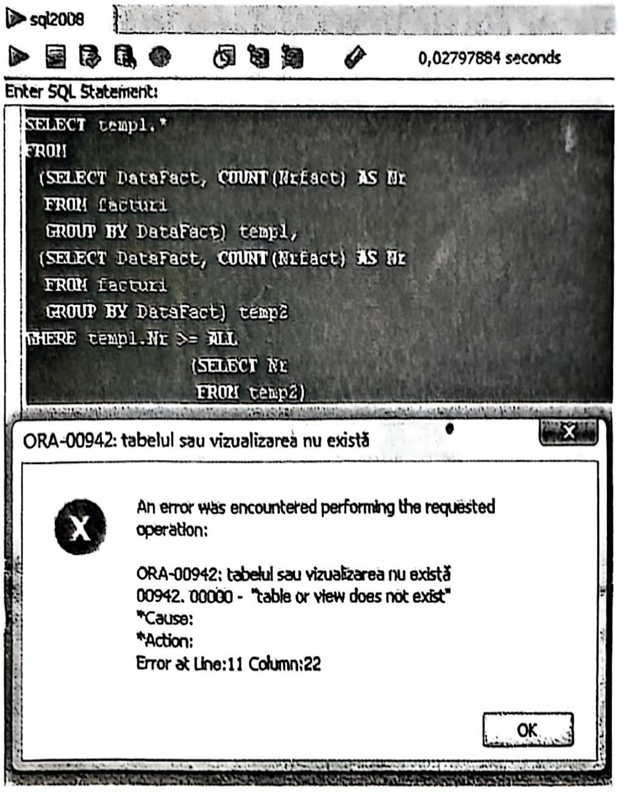


Figura 9.32. Referirea eronată a rezultatului unei subconsultări din clauza FROM

În subconsultarea din clauza HAVING a fost definită tabela „intermediară" TEMP1 al cărei atribut Nr este folosit în funcția MAX din clauza SELECT. Iar dacă tot vrem să ne complicăm viața, am putea folosi și:

```
SELECT *
FROM (SELECT DataFact, COUNT(NrFact) AS Nr
        FROM facturi
        GROUP BY DataFact) zile_fact
WHERE Nr =
        (SELECT MAX(zile_fact1.Nr)
        FROM (SELECT DataFact, COUNT(NrFact) AS Nr
                FROM facturi
                GROUP BY DataFact
                ) zile_fact1
        INNER JOIN
                (SELECT DataFact, COUNT(NrFact) AS Nr
                FROM facturi
                GROUP BY DataFact) zile_fact2
        ON zile_fact1.Nr > zile_fact2.Nr)
```

Care este clientul care a cumpărat cele mai multe produse?

```
SELECT DenCl, COUNT(DISTINCT CodPr) AS Nr_Produse
FROM clienti c
```

```
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
GROUP BY DenCl
HAVING COUNT(DISTINCT CodPr) =
        (SELECT MAX(Nr)
        FROM (SELECT CodCl, COUNT(DISTINCT CodPr) AS Nr
                FROM facturi f INNER JOIN liniifact lf
                        ON f.NrFact = lf.NrFact
                GROUP BY CodCl
                ) TEMP1)
```

Care este județul în care berea s-a vândut cel mai bine?

Avem soluții pe-alese. Iată doar două:

```
SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Bere
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE Grupa = 'Bere'
GROUP BY Judet
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) =
        (SELECT MAX(Vanzari)
        FROM (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
                FROM coduri_postale cp
                        INNER JOIN clienti c ON cp.CodPost = c.CodPost
                        INNER JOIN facturi f ON c.CodCl = f.CodCl
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                WHERE Grupa = 'Bere'
                GROUP BY Jud)                   TEMP1)
```

şi

```
SELECT bere_judete.*
FROM
        (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Bere
        FROM judete j
                INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                INNER JOIN clienti c ON cp.CodPost = c.CodPost
                INNER JOIN facturi f ON c.CodCl = f.CodCl
```

```
            INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
            INNER JOIN produse p ON lf.CodPr = p.CodPr
   WHERE Grupa = 'Bere'
  GROUP BY Judet
  ) Bere_Judete INNER JOIN
            (SELECT MAX(Vanzari) AS Max_Bere
            FROM
  (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
  FROM coduri_postale cp
            INNER JOIN clienti c ON cp.CodPost = c.CodPost
            INNER JOIN facturi f ON c.CodCl = f.CodCl
            INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
            INNER JOIN produse p ON lf.CodPr = p.CodPr
  WHERE Grupa = 'Bere'
  GROUP BY Jud)
  ) Bere_Jud_Max ON Vanzari_Bere =  Max_Bere
```

Cea de-a doua soluţie funcţionează în DB2 şi Oracle, însă în PostgreSQL şi MS SQL Server provoacă o eroare, după cum se observă în figura 9.33. Din mesaj deducem că una dintre subconsultări ar trebui să aibă un alias. Este tocmai cea pe care am pus-o în evidenţă.

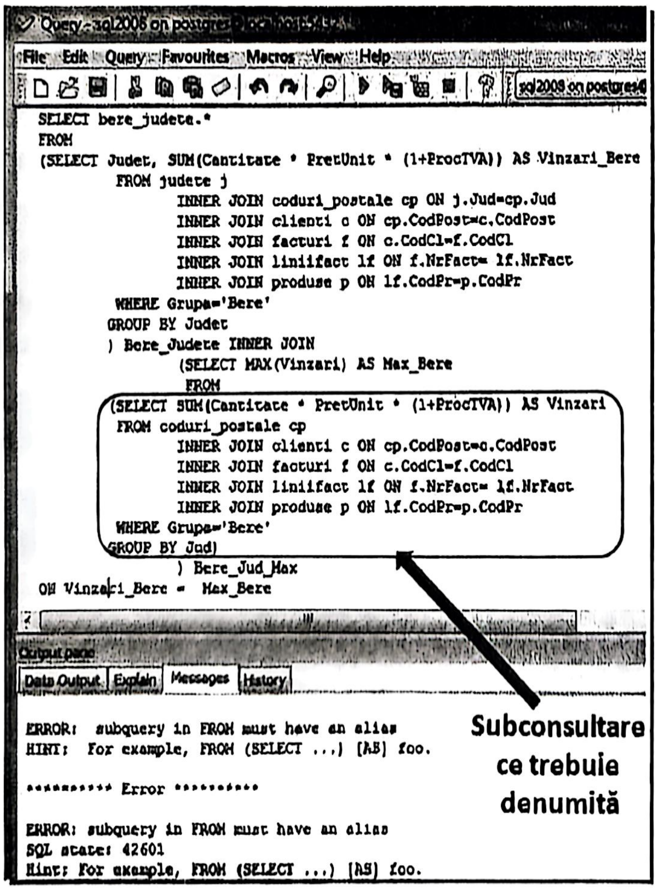


Figura 9.33. O situaţie în care aliasul unei subconsultări este obligatoriu în PostgreSQL

Acelei subconsultări îi dăm numele SUBCONSULTARE_DENUMITĂ:

```
SELECT bere_judete.*
FROM (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Bere
        FROM judete j
                INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                INNER JOIN clienti c ON cp.CodPost = c.CodPost
                INNER JOIN facturi f ON c.CodCl = f.CodCl
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE Grupa = 'Bere'
        GROUP BY Judet
        ) bere_judete INNER JOIN
                (SELECT MAX(Vanzari) AS Max_Bere
                FROM
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
        FROM coduri_postale cp
                INNER JOIN clienti c ON cp.CodPost = c.CodPost
                INNER JOIN facturi f ON c.CodCl = f.CodCl
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE Grupa = 'Bere'
        GROUP BY Jud) Subconsultare_Denumita
        ) Bere_Jud_Max
                        ON Vanzari_Bere = Max_Bere
```

şi lucrurile sunt acum în regulă. Cu atât mai mult cu cât interogarea funcţionează şi în celelalte trei servere BD. Nu ne rămâne decât să ne întrebăm de ce am dat peste eroarea aceasta aşa de târziu. Răspunsul ţine de faptul că abia acum ne-am găsit să nu denumim o subconsultare în clauza FROM.

Care sunt clienţii cu valoarea vânzărilor peste medie?

După cum am stabilit în paragraful 9.3.1, ne interesează media vânzărilor pe client calculată prin împărţirea vânzărilor totale la numărul clienţilor cărora li s-a trimis măcar o factură.

Soluţia 1:

```
SELECT DenCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenCl
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA))
```

```
        >=
        (SELECT Vanzari/NrClienti
        FROM
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
                FROM liniifact lf INNER JOIN produse p
                        ON lf.CodPr = p.CodPr
                ) temp1,
                (SELECT COUNT(DISTINCT CodCl) AS NrClienti
                FROM facturi
                ) temp2
        )
```

Soluția 2:

```
SELECT DenCl, VANZ_CL.Vanzari
FROM
        (SELECT DenCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
        FROM clienti c
                INNER JOIN facturi f ON c.CodCl=f.CodCl
                INNER JOIN liniifact lf ON f.NrFact= lf.NrFact
                INNER JOIN produse p ON lf.CodPr=p.CodPr
        GROUP BY DenCl
                ) VANZ_CL
                INNER JOIN
                (SELECT DISTINCT Vanzari/NrClienti AS Medie_Vanz
                FROM
                        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                                AS Vanzari
                        FROM liniifact lf, produse p
                        WHERE lf.CodPr = p.CodPr) X,
                                (SELECT COUNT(DISTINCT CodCl) AS NrClienti
                                FROM facturi) Y
                        ) MEDIE_VANZ
        ON VANZ_CL.Vanzari >= MEDIE_VANZ.Medie_Vanz
```

Care este factura cu cea mai mică valoare peste cea medie?

```
SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
FROM liniifact lf
        INNER JOIN produse p ON p.CodPr = lf.CodPr
GROUP BY NrFact
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) =
```

```
    (SELECT MIN(ValFact)
    FROM (
            SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
            FROM liniifact lf
                    INNER JOIN produse p ON p.CodPr = lf.CodPr
            GROUP BY NrFact
            ) TEMP1
    WHERE ValFact >
            (SELECT Vanzari / NrFacturi AS ValMedie
            FROM
                    (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                            AS Vanzari,
                    COUNT(DISTINCT NrFact) AS NrFacturi
                    FROM liniifact lf
                            INNER JOIN produse p ON p.CodPr = lf.CodPr
            ) TEMP1
            )
    )
```

Apelăm şi la o soluţie care să afişeze toate facturile cu valoarea peste medie, în ordinea crescătoare a acestei valori. Prima factură din figura 9.34 va fi răspunsul la întrebarea formulată.

```
SELECT NrFact, ValFact, ValMedie
FROM
        (SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
        FROM liniifact lf INNER JOIN produse p ON p.CodPr = lf.CodPr
        GROUP BY NrFact
        ) TEMP1
        INNER JOIN
        (SELECT ROUND(Vanzari/NrFacturi,0) AS ValMedie
        FROM
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari,
                COUNT(DISTINCT NrFact) AS NrFacturi
                FROM liniifact lf
                        INNER JOIN produse p ON p.CodPr = lf.CodPr
                ) MEDIE1
        ) MEDII
                        ON ValFact > ValMedie
ORDER BY ValFact
```

NRFACT	VALFACT	VALMEDIE
1111	4346037,5	1756083
2111	4442959,5	1756083
3111	4442959,5	1756083
1121	4737838	1756083
2121	4741271,5	1756083
1119	5774498,5	1756083
3119	5819668	1756083
2119	5819668	1756083
1114	6021706,5	1756083

Figura 9.34. Facturile cu valori peste medie

Pentru a răspunde exact la întrebarea formulată trebuie să extragem doar prima linie. Trecem sub tăcere variantele bazate pe LIMIT 1 sau TOP 1 şi ne oprim doar la una generală:

```
SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
FROM liniifact lf INNER JOIN produse p ON p.CodPr = lf.CodPr
GROUP BY NrFact
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) =
        (SELECT MIN(ValFact)
        FROM
                (SELECT NrFact, ValFact, ValMedie
                FROM
                        (SELECT NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA))
                                AS ValFact
                        FROM liniifact lf INNER JOIN produse p
                                ON p.CodPr = lf.CodPr
                        GROUP BY NrFact) TEMP1
                        INNER JOIN
                                (SELECT ROUND(Vanzari/NrFacturi,0) AS ValMedie
                                FROM
                                        (SELECT SUM(Cantitate * PretUnit *
                                                (1 + ProcTVA)) AS Vanzari,
                                                COUNT(DISTINCT NrFact)
                                                        AS NrFacturi
                                        FROM liniifact lf INNER JOIN produse p
                                                ON p.CodPr = lf.CodPr
                                        ) MEDIE1
                                ) MEDII
                                        ON ValFact > ValMedie
                ) interogarea_dinainte)
```

Care este județul cu vânzări imediat superioare județului Neamț?

```
SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY Judet
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) =
        (SELECT MIN(VANZ_JUD1.Vanzari)
        FROM
                (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
                FROM judete j
                        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                        INNER JOIN clienti c ON cp.CodPost = c.CodPost
                        INNER JOIN facturi f ON c.CodCl = f.CodCl
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY Judet
        ) VANZ_JUD1
        INNER JOIN
                (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
                FROM judete j
                        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                        INNER JOIN clienti c ON cp.CodPost = c.CodPost
                        INNER JOIN facturi f ON c.CodCl = f.CodCl
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY Judet) VANZ_JUD2
                ON VANZ_JUD1.Vanzari > VANZ_JUD2.Vanzari
                        AND VANZ_JUD2.Judet = 'Neamt')
```

Este posibil să nu vă fi revenit după trauma din paragraful 9.3.2, pricinuită de soluția problemei *Care este clientul cel mai datornic (ce are cel mai mare rest de plată)?*

De aceea, mă grăbesc în a formula o soluție care mai ameliorează depresia:

```
SELECT DenCl, Vanzari, Incasari, DeIncasat
FROM
        (SELECT FACTURAT.CodCl, Vanzari, COALESCE(Incasari, 0) AS Incasari,
                Vanzari - COALESCE(Incasari, 0) AS DeIncasat
```

```
FROM
        (SELECT CodCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY CodCl
        )       FACTURAT
                        LEFT OUTER JOIN
        (SELECT CodCl, SUM(Transa) AS Incasari
        FROM facturi f
                INNER JOIN incasfact i ON f.NrFact = i.NrFact
        GROUP BY CodCl
        ) INCASAT
                        ON FACTURAT.CodCl = INCASAT.CodCl
) TEMP1 INNER JOIN
(SELECT MAX (Vanzari - COALESCE(Incasari, 0)) AS DifMax
FROM (
        SELECT CodCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY CodCl
        ) FACTURAT LEFT OUTER JOIN
                (SELECT CodCl, SUM(Transa) AS Incasari
                FROM facturi f INNER JOIN incasfact i ON f.NrFact = i.NrFact
                GROUP BY CodCl
                ) INCASAT
        ON FACTURAT.CodCl = INCASAT.CodCl
) TEMP2
        ON TEMP1.DeIncasat = TEMP2.DifMax
INNER JOIN clienti
        ON TEMP1.CodCl = clienti.CodCl
```

În interogare obţinem o tabelă temporară cu diferenţa de încasat pe clienţi (TEMP1), şi o alta ce conţine cea mai mare diferenţă de încasat pentru un client (TEMP2). Cele două tabele sunt joncţionate după diferenţă şi, în final, pentru a afla denumirea clientului, adăugăm în joncţiune tabela CLIENŢI.

9.4.4. Diviziunea relațională *strikes back*

O parte dintre soluțiile SQL de până acum formulate pentru problemele încadrabile diviziunii relaționale (introdusă în paragraful 4.4.7) pot fi simplificate prin folosirea subconsultărilor în clauza FROM. Să începem cu succesiunea pașilor din figura 4.33). Putem adăuga cel puțin două variante la cele existente:

```
SELECT DISTINCT X
FROM rd1
WHERE X NOT IN
        (SELECT DISTINCT PRODUS_CARTEZIAN.X
        FROM (SELECT DISTINCT rd1.X, rd2.Y
                FROM rd1,rd2
                ) PRODUS_CARTEZIAN
        LEFT OUTER JOIN rd1 ON PRODUS_CARTEZIAN.X = rd1.X
        AND PRODUS_CARTEZIAN.Y = rd1.Y
        WHERE rd1.X IS NULL
        )
```

şi

```
SELECT DISTINCT X
FROM
        (SELECT X, COUNT(Y) AS Nr
        FROM rd1
        GROUP BY X
        ) TEMP1 INNER JOIN
        (SELECT COUNT(Y) AS Nr
        FROM rd2
        ) TEMP2 ON TEMP1.Nr = TEMP2.Nr
```

În ultima interogare, tabela temporară ad-hoc TEMP1 conține numărul valorilor lui Y pentru fiecare X din R1, iar TEMP2, numai numărul total al valorilor lui Y din R2 - figura 9.35.

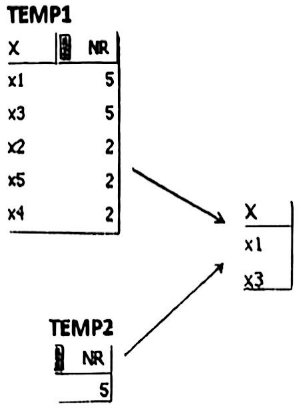


Figura 9.35. Tabelele temporare TEMP1 şi TEMP2

Ce produse au fost vândute tuturor clienţilor? (paragraful 4.4.7, exemplul 26)

Dintre cele două variante de lucru propuse la începutul acestui paragraf, apelăm (probabil, pe nedrept) la cea de-a doua:

```
SELECT DISTINCT DenPr
FROM (SELECT DenPr, COUNT(DISTINCT CodCl) AS Nr
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        GROUP BY DenPr
        ) TEMP1 INNER JOIN (SELECT COUNT(CodCl) AS Nr FROM clienti) TEMP2
                ON TEMP1.Nr = TEMP2.Nr
```

În ce zile s-au vândut şi produsul cu denumirea „Produs 1", şi cel cu denumirea „Produs 2"? (exemplul 23, paragraful 4.4.7)

```
SELECT DISTINCT DataFact
FROM
        (SELECT F.DataFact, COUNT(DISTINCT LF.CodPr) AS Nr
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE DenPr IN ('Produs 1', 'Produs 2')
        GROUP BY F.DataFact) TEMP1
WHERE Nr = 2
```

TEMP1 conţine, pentru fiecare dată calendaristică, numărul de produse, dintre Produs 1 şi Produs 2 (1 sau 2) care au fost vândute în ziua respectivă.

Care sunt clienţii pentru care există cel puţin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007? (exemplul 23, paragraful 4.4.7)

Soluţia 1:

```
SELECT DISTINCT DenCl
FROM (
        SELECT clienti.*
        FROM clienti INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        ) un_alias
WHERE CodCl NOT IN
        (SELECT DISTINCT PRODUS_CARTEZIAN.CodCl
        FROM (
                SELECT DISTINCT rd1.CodCl, rd2.DataFact
                FROM (
                        SELECT *
                        FROM facturi
```

```
                    WHERE DataFact BETWEEN DATE'2007-09-10'
                                    AND DATE'2007-09-30'
            ) rd1 CROSS JOIN (SELECT DISTINCT DataFact
                    FROM facturi
                    WHERE DataFact BETWEEN
                                            DATE'2007-09-10' AND
                                            DATE'2007-09-30') rd2
                                    ) PRODUS_CARTEZIAN
            LEFT OUTER JOIN (SELECT CodCl, DataFact
            FROM facturi
            WHERE DataFact BETWEEN DATE'2007-09-10'
                                    AND DATE'2007-09-30' ) rd1
            ON PRODUS_CARTEZIAN.CodCl = rd1.CodCl AND
                            PRODUS_CARTEZIAN.DataFact = rd1.DataFact
            WHERE rd1.CodCl IS NULL
        )
```

Soluția 2:

```
SELECT DISTINCT DenCl
FROM
        (SELECT CodCl, COUNT(DISTINCT DataFact) AS Nr
        FROM facturi
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        GROUP BY CodCl
        ) TEMP1
                INNER JOIN
                        (SELECT COUNT(DISTINCT DataFact) AS Nr
                        FROM facturi
                        WHERE DataFact BETWEEN DATE'2007-09-10'
                                AND DATE'2007-09-30'
                        ) TEMP2
                                ON TEMP1.Nr = TEMP2.Nr
                INNER JOIN clienti
                                ON TEMP1.CodCl = clienti.CodCl
```

Ce facturi conțin măcar produsele din factura 1117? (exemplul 27, paragraful 4.4.7)

```
SELECT DISTINCT NrFact
FROM
        (
        SELECT NrFact, COUNT(*) AS NrProd
        FROM liniifact
```

```
WHERE CodPr IN
        (SELECT CodPr
        FROM liniifact
        WHERE NrFact = 1117)
GROUP BY NrFact
) T1 INNER JOIN
        (SELECT COUNT(CodPr) AS NrP1117
        FROM liniifact
        WHERE NrFact = 1117
                ) T2
ON T1.NrProd = T2.NrP1117
```

T2 conţine numărul produselor din factura 1117. T1 conţine, pentru fiecare factură, câte produse sunt comune acesteia şi facturii 1117. Prin joncţiunea tabelelor „ad-hoc" T1 şi T2 prin condiţia *T1.NrProd = T2.NrP1117*, se extrag acele linii din T1 care au acelaşi număr de produse prezente în factura 1117 ca şi aceasta (adică factura 1117).

9.4.5. Tabele temporare create ad-hoc

În paragraful 7.3 am formulat o problemă care viza numărul de produse vândute, pe intervale ale preţurilor unitare de ordinul miilor. Schimbăm enunţul astfel:

Să se afle numărul de produse vândute, pe următoarele intervale ale preţului unitar de vânzare:

- *între 0 şi 500 RON;*
- *între 501 şi 750 RON;*
- *între 751 şi 1000 RON;*
- *între 1001 şi 1500 RON;*
- *între 1501 şi 5000 RON;*
- *între 5001 şi 6000 RON;*
- *între 6001 şi 7000 RON;*
- *peste 7001 RON.*

Intervalele sunt, după cum vedem, neregulate, astfel încât artificiile care ar utiliza funcţii TRUNC/ROUND sunt mai greu de folosit. Beneficiind de minunăţia includerii subconsultărilor în clauza FROM, putem încerca o variantă intermediară în Oracle astfel:

```
SELECT *
FROM
        (SELECT 0 AS LimInf, 500 AS LimSup FROM dual UNION
                SELECT 501, 750 FROM dual UNION
                SELECT 751, 1000 FROM dual UNION
                SELECT 1001, 5000 FROM dual UNION
                SELECT 5001, 6000 FROM dual UNION
```

```
                    SELECT 6001, 7000 FROM dual UNION
                    SELECT 7001, 99999999 FROM dual
        ) intervale
        LEFT OUTER JOIN liniifact ON PretUnit BETWEEN LimInf AND LimSup
ORDER BY LimInf, LimSup, PretUnit
```

Fiecare interval este definit printr-un SELECT în care apar două coloane - limitele inferioare (LimInf) şi superioare (LimSup) ale acestuia. Cele şapte SELECT-uri sunt conectate prin operatorul UNION, tabela temporară ad-hoc INTERVALE fiind cea din figura 9.36.

LIMINF	LIMSUP
0	500
501	750
751	1000
1001	5000
5001	6000
6001	7000
7001	99999999

Figura 9.36. Tabelele temporară (ad-hoc) INTERVALE

Această tabelă este (theta-)joncţionată extern cu tabela LINIIFACT, în vederea încadrării fiecărui preţ unitar în intervalul corespunzător - vezi figura 9.37 (figură care conţine doar o parte din liniile rezultatului).

LIMINF	LIMSUP	NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
0	500	(null)	(null)	(null)	(null)	(null)
501	750	2119	2	3	40	700
501	750	3119	2	3	40	700
501	750	1119	2	3	40	700
501	750	2112	2	3	65	750
501	750	1112	2	3	40	750
501	750	3112	2	3	65	750
751	1000	2115	1	2	110	925
751	1000	1115	1	2	150	925
751	1000	3115	1	2	110	925
751	1000	1116	1	2	125	930
751	1000	3118	2	1	120	930
751	1000	1118	2	1	150	930
751	1000	2118	2	1	120	930
751	1000	3116	1	2	[illegible]	930

Figura 9.37. Încadrarea preţurilor unitare pe intervale (fragment)

În continuare, nu ne rămâne decât să facem gruparea după intervale. Argumentul funcţiei COUNT trebuie să fie, neapărat, un atribut din tabela

LINIIFACT, pentru că, altminteri, la intervalele fără nici un preţ ar apărea eronat valoarea 1:

```
SELECT LimInf, LimSup, COUNT(liniifact.PretUnit) AS Nr_produse
FROM
        (SELECT 0 AS LimInf, 500 AS LimSup FROM dual UNION
          SELECT 501, 750 FROM dual UNION
           SELECT 751, 1000 FROM dual UNION
            SELECT 1001, 5000 FROM dual UNION
             SELECT 5001, 6000 FROM dual UNION
              SELECT 6001, 7000 FROM dual UNION
               SELECT 7001, 99999999 FROM dual
        ) intervale
      LEFT OUTER JOIN liniifact ON PretUnit BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

Figura 9.38 conţine informaţiile solicitate.

LIMINF	LIMSUP	NR_PRODUSE
0	500	0
501	750	6
751	1000	21
1001	5000	20
5001	6000	0
6001	7000	3
7001	99999999	6

Figura 9.38. Numărul corect de preţuri unitare, pe fiecare interval

În DB2 trebuie nu numai ca DUAL să fie înlocuit cu SYSIBM.SYSDUMMY1 (sau SYSIBM.DUAL), dar în toate cele şapte SELECT-uri conectate prin UNION trebuie folosită clauza AS pentru ca atributele să „păstreze" titulaturile *LimInf* şi *LimSup*:

```
SELECT LimInf, LimSup, COUNT(liniifact.PretUnit) AS Nr_produse
FROM
        (SELECT 0 AS LimInf, 500 AS LimSup FROM sysibm.sysdummy1 UNION
         SELECT 501 AS LimInf, 750 AS LimSup FROM sysibm.sysdummy1 UNION
          SELECT 751 AS LimInf, 1000 AS LimSup FROM sysibm.sysdummy1 UNION
           SELECT 1001 AS LimInf, 5000 AS LimSup FROM sysibm.sysdummy1 UNION
               SELECT 5001 AS LimInf, 6000 AS LimSup FROM sysibm.sysdummy1
         UNION SELECT 6001 AS LimInf, 7000 AS LimSup FROM sysibm.sysdummy1
UNION SELECT 7001 AS LimInf, 99999999 AS LimSup FROM sysibm.sysdummy1
         ) intervale
                LEFT OUTER JOIN liniifact
```

```
        ON PretUnit BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

Pentru portarea soluției în PostgreSQL şi MS SQL Server nu ne rămâne decât ca din sintaxa Oracle să eliminăm *FROM dual*. PostgreSQL-ul şi DB2-ul pun la dispoziție, însă, şi o soluție bazată pe folosirea clauzei VALUES pentru a defini o „tabelă-constantă" (INTERVALE) chiar în clauza FROM, ceea ce economiseşte mult din lungimea interogării (vezi şi paragrafele 3.4.4 şi 3.4.6):

```
SELECT LimInf, LimSup, COUNT(liniifact.PretUnit) AS Nr_produse
FROM
        (VALUES (0, 500), (501, 750), (751, 1000), (1001, 5000),
        (5001, 6000), (6001, 7000), (7001, 99999999)
        ) AS intervale (LimInf, LimSup)
                LEFT OUTER JOIN liniifact
        ON PretUnit BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

Să se afle numărul de facturi emise, pe următoarele intervale ale valorilor totale:

- *între 0 şi 100000 RON;*
- *între 100001 şi 200000 RON;*
- *între 200001 şi 500000 RON;*
- *între 500001 şi 1000000 RON;*
- *peste 1000001 RON.*

Singura diferență majoră față problema anterioară ține de faptul că valoarea unei facturi nu este un atribut la nivel de tuplu (cum era prețul unitar), ci un atribut la nivel de grup. Persistăm în soluții oracliste:

```
SELECT LimInf, LimSup, COUNT(facturi.ValFact) AS Nr_Facturi
FROM
  (SELECT 0 AS LimInf, 100000 AS LimSup FROM dual UNION
   SELECT 100001, 200000 FROM dual UNION
    SELECT 200001, 500000 FROM dual UNION
     SELECT 500001, 1000000 FROM dual UNION
        SELECT 1000001, 99999999 FROM dual
     ) intervale
  LEFT OUTER JOIN
        (SELECT f.NrFact,
                SUM(Cantitate * PretUnit * (1+ProcTVA)) AS ValFact
        FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY f.NrFact
```

```
        ) facturi
                        ON ValFact BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

LIMINF	LIMSUP	NR_FACTURI
0	100000	1
100001	200000	14
200001	500000	4
500001	1000000	0
1000001	99999999	9

Figura 9.39. Numărul facturi, pe intervale de valori

În PostgreSQL şi SQL Server trebuie doar să eliminăm *FROM dual*-urile; în DB2 *dual*-urile trebuie înlocuite cu *sysibm.DUAL*-uri, iar clauza AS trebuie folosită pentru coloanele tuturor SELECT-urilor - interval. Ca bonus, acelaşi rezultat se obţine în DB2 şi PostgreSQL fie prin înlocuirea/eliminarea din interogarea de mai sus a secvenţelor *FROM dual* (obţinem, astfel, şi versiunea SQL Server), fie cu ajutorul clauzei VALUES folosită ca în precedentul exemplu:

```
SELECT LimInf, LimSup, COUNT(facturi.ValFact) AS Nr_Facturi
FROM
        (VALUES (0, 100000) , (100001, 200000), (200001, 500000),
        (500001, 1000000), (1000001, 99999999)
        ) intervale (LimInf, LimSup)
LEFT OUTER JOIN
        (SELECT f.NrFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
        FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY f.NrFact
        ) facturi
                ON ValFact BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

Care sunt vânzările pentru toate cele zece zile cuprinse în intervalul 1-10 septembrie 2007?

De data aceasta, se doreşte includerea în rezultat şi a zilelor fără nicio factură întocmită, ca în figura 9.40.

Soluţiile se redactează pe calapodul precedentelor, astfel:

- în Oracle:

```
SELECT Zi, TRUNC(COALESCE(SUM(facturi.ValFact),0)) AS Nr_Facturi
```

```
FROM
  (SELECT DATE'2007-09-01' AS Zi FROM dual UNION
   SELECT DATE'2007-09-02' FROM dual UNION
    SELECT DATE'2007-09-03' FROM dual UNION
     SELECT DATE'2007-09-04' FROM dual UNION
      SELECT DATE'2007-09-05' FROM dual UNION
       SELECT DATE'2007-09-06' FROM dual UNION
        SELECT DATE'2007-09-07' FROM dual UNION
         SELECT DATE'2007-09-08' FROM dual UNION
          SELECT DATE'2007-09-09' FROM dual UNION
           SELECT DATE'2007-09-10' FROM dual
    ) zile
 LEFT OUTER JOIN
    (SELECT f.NrFact, DataFact,
       SUM(Cantitate * PretUnit * (1+ProcTVA)) AS ValFact
     FROM facturi f INNER JOIN liniifact lf ON lf.NrFact=f.NrFact
       INNER JOIN produse p ON lf.CodPr=p.CodPr
     GROUP BY f.NrFact, DataFact ) facturi
    ON Zi=TRUNC(DataFact)
GROUP BY Zi
ORDER BY Zi
```

ZI	NR_FACTURI
01-09-2007	4596401
02-09-2007	238437
03-09-2007	0
04-09-2007	0
05-09-2007	0
06-09-2007	0
07-09-2007	0
08-09-2007	0
09-09-2007	0
10-09-2007	424704

Figura 9.40. Vânzările din fiecare zi din intervalul 1-10 sept. 2007

- în MS SQL Server:

```
SELECT Zi, ROUND(COALESCE(SUM(facturi.ValFact),0),0,1) AS Nr_Facturi
FROM
  (SELECT '2007-09-01' AS Zi  UNION  SELECT '2007-09-02'  UNION
    SELECT '2007-09-03'  UNION SELECT '2007-09-04'  UNION
      SELECT '2007-09-05'  UNION SELECT '2007-09-06'  UNION
```

```
        SELECT '2007-09-07' UNION SELECT '2007-09-08' UNION
          SELECT '2007-09-09' UNION SELECT '2007-09-10'
      ) zile
  LEFT OUTER JOIN
    (SELECT f.NrFact, DataFact,
      SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
     FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
      INNER JOIN produse p ON lf.CodPr = p.CodPr
     GROUP BY f.NrFact, DataFact ) facturi
    ON Zi = DataFact
GROUP BY Zi
ORDER BY Zi
```

- în DB2/PostgreSQL:

```
SELECT Zi, TRUNC(COALESCE(SUM(facturi.ValFact),0),0) AS Nr_Facturi
FROM (VALUES
     (CAST ('2007-09-01' AS DATE)), (CAST ('2007-09-02' AS DATE)),
     (CAST ('2007-09-03' AS DATE)), (CAST ('2007-09-04' AS DATE)),
     (CAST ('2007-09-05' AS DATE)), (CAST ('2007-09-06' AS DATE)),
     (CAST ('2007-09-07' AS DATE)), (CAST ('2007-09-08' AS DATE)),
     (CAST ('2007-09-09' AS DATE)), (CAST ('2007-09-10' AS DATE))
        ) AS zile (Zi)
     LEFT OUTER JOIN
        (SELECT f.NrFact, DataFact,
                SUM(Cantitate * PretUnit * (1+ProcTVA)) AS ValFact
        FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY f.NrFact, DataFact ) facturi
                        ON Zi = DataFact
     GROUP BY Zi
     ORDER BY Zi
```

Tot în PostgreSQL există funcția-tabelă GENERATE_SERIES care creează un set de linii corespunzătoare unui interval şi unui pas/increment. De exemplu, dacă dorim să generăm o tabelă în care fiecare linie să conțină o zi din luna septembrie 2007, se poate folosi interogarea:

```
SELECT DATE'2007-09-01' + Nr AS Zi
FROM GENERATE_SERIES(0,29,1) AS serie (Nr)
```

În aceste condiții, putem simplifica (oarecum) interogarea PostgreSQL pentru problema noastră:

```
SELECT Zi, TRUNC(COALESCE(SUM(facturi.ValFact),0)) AS Nr_Facturi
```

```
FROM
        (SELECT DATE'2007-09-01' + Nr AS Zi
        FROM GENERATE_SERIES(0,9,1) Serie (Nr)
        ) Zile
                LEFT OUTER JOIN
        (SELECT f.NrFact, DataFact,
                SUM(Cantitate * PretUnit * (1+ProcTVA)) AS ValFact
        FROM facturi f
                        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY f.NrFact, DataFact ) facturi
                                ON Zi = DataFact
GROUP BY Zi
ORDER BY Zi
```

În paragraful 8.4 am făcut cunoştinţă cu o funcţie proprietară PostgreSQL, CROSSTAB. O folosim din nou, la o problemă care să pună în valoare şi GENERATE_SERIES. Dorim să comparăm, vânzările din primele 7 zile pentru lunile august şi septembrie 2007, ca în figura 9.41.

luna integer	01/luna/2007 numeric	02/luna/2007 numeric	03/luna/2007 numeric	04/luna/2007 numeric	05/luna/2007 numeric	06/luna/2007 numeric	07/luna/2007 numeric
8	10599536.0000	277950.0000	222050.0000	201975.0000			10610000.5000
9	4596401.5000	238437.5000					

Figura 9.41. Folosirea funcţiilor GENERATE_SERIES şi CROSSTAB

Interogarea PostgreSQL se prezintă astfel:

```
SELECT *
FROM crosstab(
        'SELECT EXTRACT (MONTH FROM DataFact) AS Luna,
            EXTRACT (DAY FROM DataFact) AS Ziua, Vanzari
        FROM (
            SELECT DataFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
            FROM facturi
                INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
                INNER JOIN produse ON liniifact.CodPr = produse.CodPr
            WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) BETWEEN 8 AND 9 AND
                EXTRACT (DAY FROM DataFact) BETWEEN 1 AND 7
            GROUP BY DataFact
                ) x
        ORDER BY 1,2',
        'SELECT Nr FROM generate_series (1,7) Nr '
```

```
        ) AS (
        Luna INT,
        "01/luna/2007" NUMERIC,
        "02/luna/2007" NUMERIC,
        "03/luna/2007" NUMERIC,
        "04/luna/2007" NUMERIC,
        "05/luna/2007" NUMERIC,
        "06/luna/2007" NUMERIC,
        "07/luna/2007" NUMERIC
);
```

9.5. Subconsultări scalare în clauza SELECT

Subconsultările scalare sunt cele care obţin un rezultat alcătuit dintr-o singură linie şi o singură coloană. Dacă subconsultările incluse în clauzele WHERE, HAVING şi FROM puteau obţine un rezultat tabelar oarecare, în clauza SELECT suntem constrânşi să includem doar interogări scalare.

Care sunt totalurile vânzărilor şi încasărilor?

Ca de obicei, începem cu o problemă slabă, pentru care formulăm o soluţie curioasă, mai întâi în PostgreSQL/SQL Server:

```
SELECT
    (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari
    FROM liniifact lf INNER JOIN produse p
        ON lf.CodPr = p.CodPr
        ) AS Facturat,
    (SELECT SUM (Transa) FROM incasfact) AS Incasat
```

apoi echivalenta sa în Oracle:

```
SELECT
    ...
FROM dual
```

şi cea din DB2:

```
SELECT
    ...
FROM SYSIBM.SYSDUMMY1
```

Fraza SELECT conţine două subconsultări scalare, una care extrage totalul vânzărilor, iar cealaltă, totalul încasărilor. În PostgreSQL şi SQL Server clauza FROM poate lipsi, în timp ce în Oracle nu, aşa că, fiind nevoie de o tabelă cu o singură linie, s-a folosit tabela DUAL (creată automat în Oracle la instalare) şi SYSIBM.SYSDUMMY1 care, după cum am văzut încă din capitolul 5, are o singură

linie şi o singură coloană. Indiferent de variantă, cele două rezultate ale interogărilor scalare au valorile din figura 9.42.

FACTURAT	INCASAT
49170335	518281

Figura 9.42. Două interogări scalare în clauza SELECT

Care sunt totalurile salariilor tarifare ale angajaţilor şi ale sporurilor pe luna iulie 2007 pentru întreaga firmă?

Atributul SalTarifar se află în tabela PERSONAL2, iar sporurile în tabela SPORURI. Până în acest paragraf am putea folosi câteva variante, precum:

```
SELECT SUM(SalTarifar) AS Total_Sal_Tarifar, SUM ( COALESCE(SporVechime,0) +
        COALESCE(SporNoapte,0) + COALESCE(SporCD,0) + COALESCE(AlteSpor,0))
            AS Total_Sporuri_Iulie
FROM personal2 INNER JOIN sporuri
ON personal2.Marca = sporuri.Marca WHERE An=2007 AND Luna = 7
```

sau

```
SELECT *
FROM (
        SELECT SUM(SalTarifar) AS Total_Sal_Tarifar
        FROM personal2
        ) un_total,
        (SELECT SUM ( COALESCE(SporVechime,0) + COALESCE(SporNoapte,0)
            + COALESCE(SporCD,0) + COALESCE(AlteSpor,0))
            AS Total_Sal_Tarifar
        FROM sporuri
        WHERE An = 2007 AND Luna = 7
            ) alt_total
```

Fiind vorba despre două valori scalare, putem include două interogări în clauza SELECT, iar în clauza FROM a interogării principale folosim, după caz, DUAL (în Oracle), SYSIBM.SYSDUMMY1 (în DB2) sau nimic (în PostgreSQL şi MS SQL Server):

```
SELECT (
        SELECT SUM(SalTarifar) FROM personal2
        ) AS Total_Sal_Tarifar,
        (
        SELECT SUM ( COALESCE(SporVechime,0) +
            COALESCE(SporNoapte,0) + COALESCE(SporCD,0) +
            COALESCE(AlteSpor,0) )
        FROM sporuri
```

```
        WHERE An = 2007 AND Luna = 7
) AS Total_Sal_Tarifar
FROM dual
```

Curios lucru, interogarea:

```
SELECT SUM(SalTarifar) AS Total_Sal_Tarifar,
        (SELECT SUM ( COALESCE(SporVechime,0) +
                COALESCE(SporNoapte,0) +
                COALESCE(SporCD,0) + COALESCE(AlteSpor,0))
        FROM sporuri
        WHERE An = 2007 AND Luna = 7) AS Total_Sporuri_Iulie
FROM personal2
```

funcţionează în DB2, PostgreSQL şi SQL Server, dar nu şi în Oracle (vezi figura 9.43).

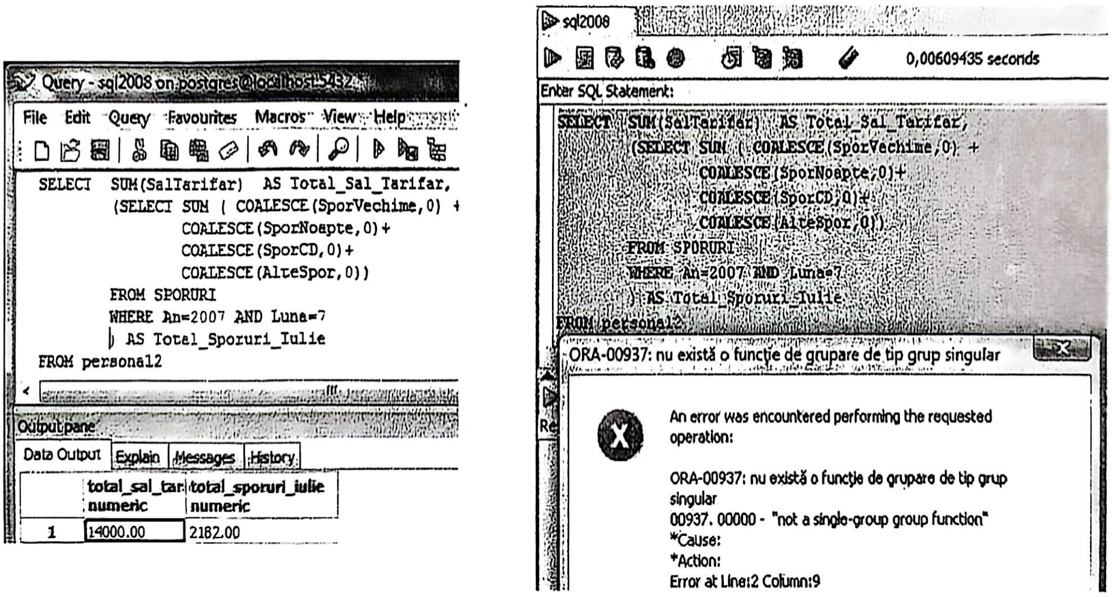


Figura 9.43. O interogare funcţionabilă în PostgreSQL şi SQL Server, dar nu şi în Oracle

Pentru cât la sută dintre clienţi s-au întocmit, zilnic, facturi?

```
SELECT DataFact, COUNT(DISTINCT CodCl) AS Nr_Clienti,
        (SELECT COUNT(*)
        FROM clienti
        ) AS Nr_Total_Clienti,
        ROUND((COUNT(DISTINCT CodCl) * 100)/(
        SELECT COUNT(*)
        FROM clienti)
,0) AS Procent
FROM facturi
GROUP BY DataFact
ORDER BY 1
```

Pentru a obține procentul care ne interesează se împarte rezultatul calculat de funcția COUNT pentru fiecare grup (zi calendaristică) la valoarea extrasă de interogarea scalară (vezi figura 9.44). Soluția este valabilă pentru toate cele patru dialecte.

DataFact	Nr_Clienti	Nr_Total_Clienti	Procent
2007-08-01 00:00:00	4	7	57
2007-08-02 00:00:00	2	7	28
2007-08-03 00:00:00	1	7	14
2007-08-04 00:00:00	1	7	14
2007-08-07 00:00:00	4	7	57
2007-08-14 00:00:00	3	7	42
2007-08-15 00:00:00	2	7	28
2007-08-16 00:00:00	1	7	14
2007-08-21 00:00:00	2	7	28
2007-08-22 00:00:00	1	7	14
2007-09-01 00:00:00	2	7	28
2007-09-02 00:00:00	2	7	28
2007-09-10 00:00:00	2	7	28
2007-09-17 00:00:00	1	7	14
2007-10-07 00:00:00	1	7	14

Figura 9.44. Procentajul zilnic al clienților pentru care există facturi

Care este contribuția (procentuală) a fiecărui produs la totalul vânzărilor?

Informația este esențială pentru orice firmă. Ne vom servi de o subconsultare scalară pentru a determina, pe fiecare linie (corespunzătoare unui produs) totalul vânzărilor și a obține, astfel, raportul care ne interesează[5].

```
SELECT DenPr AS Produs,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Produs,
        (
        SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
        FROM liniifact lf
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        ) AS Total_Vanzari,
        TRUNC(
                (SUM(Cantitate * PretUnit * (1 + ProcTVA)) * 100)/
                        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                        FROM liniifact lf INNER JOIN produse p
                                ON lf.CodPr = p.CodPr
                        )
                        , 2 ) AS Procent
  FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
```

5. Cele patru servere BD prezintă diferențe în modul de afișare a pozițiilor fracționare. Figura redă rezultatul din PostgreSQL.

```
GROUP BY DenPr
ORDER BY 1
```

produs character var	vinzari_produ numeric	total_vinzari numeric	procent numeric
Produs 1	988533.0000	49170335.0000	2.01
Produs 2	2784023.5000	49170335.0000	5.66
Produs 3	251685.0000	49170335.0000	0.51
Produs 4	301657.5000	49170335.0000	0.61
Produs 5	44844436.0000	49170335.0000	91.20

Figura 9.45. Contribuţia fiecărui produs la cifra de afaceri

Care este ziua în care s-au emis cele mai multe facturi?

```
SELECT DataFact, COUNT(*) AS Nr_Facturi,
        (SELECT MAX(NrF)
        FROM
                (SELECT COUNT(*) AS NrF
                 FROM facturi
                GROUP BY DataFact) T1 ) AS NrMax
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) >=
        (SELECT MAX(NrF)
        FROM
                (SELECT COUNT(*) AS NrF
                 FROM facturi
                GROUP BY DataFact) T2 )
```

Valorile coloanelor Nr_Facturi şi NrMax din figura 9.46 sunt furnizate de două subconsultări scalare, una care calculează numărul zilnic al facturilor, iar a doua numărul maxim de facturi emise într-o zi.

DATAFACT	NR_FACTURI	NRMAX
01-08-2007	4	4
07-08-2007	4	4

Figura 9.46. Zilele în care s-au întocmit cele mai multe facturi

Din păcate, coloana calculată printr-o interogare scalară nu poate fi folosită în alte clauze (WHERE, HAVING), nici măcar în aceaşi clauză SELECT. Astfel, ne-am fi imaginat că la problema *Ce facturi au fost emise în aceeaşi zi cu factura 1120?* am putea formula şi o soluţie de genul:

```
SELECT facturi.*,
        (SELECT DataFact FROM facturi WHERE NrFact = 1120
        ) AS Data_F_1120
```

FROM facturi

WHERE DataFact = Data_F_1120

Aceasta ar fi vestea rea (vezi figura 9.47). Vestea bună este că subconsultările scalare din clauza SELECT pot fi corelate cu tabelele din clauza FROM, după cum vom vedea în capitolul următor.

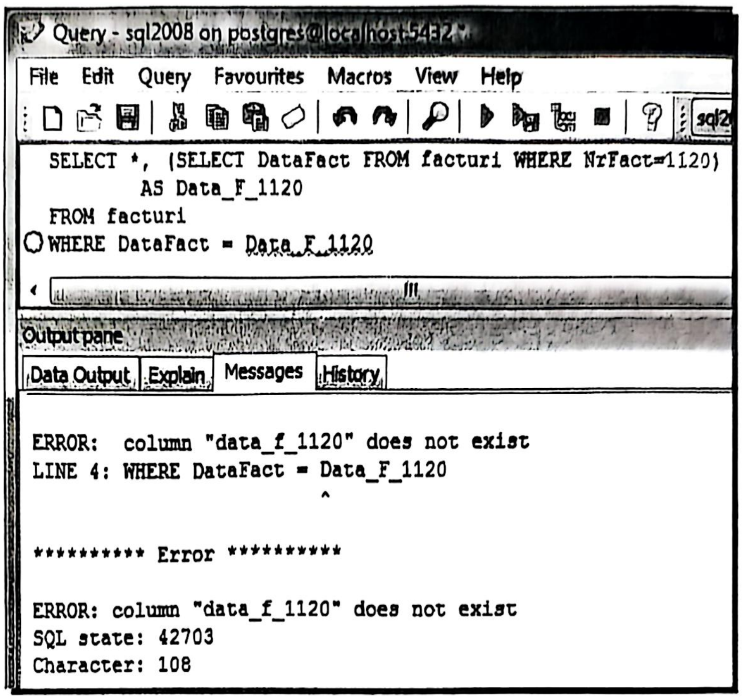


Figura 9.47. Subconsultarea din clauza SELECT nu este „recunoscută" în celelalte clauze ale interogării

9.6. Expresii-tabelă

În prima ediție a cărții prezentam eleganta opțiune a expresiilor-tabele doar în DB2. Între timp și Oracle și SQL Server le-au implementat. Revenim (pentru a treia oară) la exemplul 19 din algebra relațională: *Ce facturi au fost emise în aceeași zi cu factura 1120?*

Iată și soluția DB2/Oracle/SQL Server bazată pe expresii-tabele:

```
WITH Zi_1120 AS
        (SELECT DataFact
        FROM facturi
        WHERE NrFact = 1120)
SELECT NrFact
FROM facturi, Zi_1120
WHERE facturi.DataFact = Zi_1120.DataFact
```

ZI_1120 este o expresie-tabelă cu o singură coloană - DataFact - și o singură linie care conține data întocmirii facturii 1120, fiind folosită în fraza SELECT principală ca o tabelă obișnuită. Aici e marele avantaj, prin comparație cu

subconsultările din clauza FROM: tabela-expresie este recunoscută în toate clauzele şi subconsultările „implicate" în interogarea principală şi subconsultările ei. Continuăm cu trei probleme de „maxim" rezolvate cu noua găselniţă.

Care este ziua în care s-au emis cele mai multe facturi?

```
WITH facturi_zile AS
        (SELECT DataFact, COUNT(*) AS Nr_Facturilor
        FROM facturi
        GROUP BY DataFact)
SELECT *
FROM facturi_zile
WHERE Nr_Facturilor =
    (SELECT MAX(Nr_Facturilor)
FROM facturi_zile)
```

Care este clientul care a cumpărat cele mai multe produse?

```
WITH clienti_produse AS
        (SELECT DenCl, COUNT(DISTINCT CodPr) AS Nr_Prod
        FROM clienti c
            INNER JOIN facturi f ON c.CodCl = f.CodCl
            INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        GROUP BY DenCl)
SELECT *
FROM clienti_produse
WHERE Nr_Prod =
    (SELECT MAX(Nr_Prod)
    FROM clienti_produse)
```

Care este judeţul în care berea s-a vândut cel mai bine?

```
WITH judete_bere AS
        (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vnz_Bere
        FROM judete j
            INNER JOIN coduri_postale cp ON j.Jud=cp.Jud
            INNER JOIN clienti c ON cp.CodPost = c.CodPost
            INNER JOIN facturi f ON c.CodCl = f.CodCl
            INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
            INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE Grupa = 'Bere'
        GROUP BY Judet)
SELECT *
```

```
FROM judete_bere
WHERE Vnz_Bere =
        (SELECT MAX(Vnz_Bere)
        FROM judete_bere)
```

Într-o interogare putem declara două sau mai multe expresii-tabelă pe care să le folosim în toate clauzele, în aceeaşi manieră în care folosim celelalte tabele ale bazei de date. Să luăm un exemplu folosit de-a lungul şi de-a latul ultimelor capitole pentru ilustrarea atât a intersecţiei, cât şi a diviziunii:

În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"?

De această dată au fost definite două expresii-tabelă, ZILE_PRODUS1 ce conţine zilele în care s-a vândut produsul 1, şi ZILE_PRODUS2 care „adăposteşte" zilele în care s-a vândut cel de-al doilea produs. Fraza SELECT principală îndeplineşte o simplă formalitate - intersecţia celor două tabele ad-hoc:

```
WITH zile_produs1 AS
            (SELECT DISTINCT DataFact
            FROM facturi f
                    INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
            WHERE DenPr = 'Produs 1'),
        zile_produs2 AS
            (SELECT DISTINCT DataFact
            FROM facturi f
                    INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
            WHERE DenPr = 'Produs 2')
SELECT *
FROM zile_produs1
        INTERSECT
SELECT *
FROM zile_produs2
```

Interogările din paragraful 9.4.5 - dedicat tabelelor temporare ad-hoc declarate în clauza FROM - pot fi, uneori, vizibil simplificate. Să luăm doar unul dintre aceste exemple, deşi economia interogării nu se ameliorează spectaculos. *Să se afle numărul de produse vândute, pe următoarele intervale ale preţului unitar de vânzare: între 0 şi 500 RON; între 501 şi 750 RON; între 751 şi 1000 RON... peste 7001 RON.* Enunţul este cel al primului exemplu din paragraful 9.4.5 (vezi figura 9.38). Iată soluţia bazată pe expresii-tabelă, redactată după sintaxa SQL Server:

```
WITH intervale AS
(SELECT 0 AS LimInf, 500 AS LimSup UNION
        SELECT 501, 750 UNION
```

```
        SELECT 751, 1000 UNION
          SELECT 1001, 5000 UNION
            SELECT 5001, 6000 UNION
              SELECT 6001, 7000 UNION
                SELECT 7001, 99999999
    )
SELECT LimInf, LimSup, COUNT(liniifact.PretUnit) AS Nr_produse
FROM intervale
    LEFT OUTER JOIN liniifact ON PretUnit BETWEEN LimInf AND LimSup
GROUP BY LimInf, LimSup
ORDER BY LimInf, LimSup
```

Ce produse au fost vândute tuturor clienţilor? (paragraful 4.4.7, exemplul 26)

```
WITH produse_clienti AS
        (SELECT DenPr, COUNT(DISTINCT CodCl) AS Nr
        FROM produse p
            INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
            INNER JOIN facturi f ON lf.NrFact = f.NrFact
        GROUP BY DenPr),
    nr_clienti AS
        (SELECT COUNT(CodCl) AS NrClienti
        FROM clienti
        )
SELECT DenPr
FROM produse_clienti INNER JOIN nr_clienti ON Nr = NrClienti
```

Care sunt clienţii pentru care există cel puţin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007? (exemplu 23, paragraf 4.4.7)

```
WITH clienti_zile AS (
        SELECT DenCl, COUNT(DISTINCT DataFact) AS NrZileCl
        FROM facturi INNER JOIN clienti
            ON facturi.CodCl = clienti.CodCl
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        GROUP BY DenCl
        ),
    zile AS (
        SELECT COUNT(DISTINCT DataFact) AS NrZile
        FROM facturi
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        )
```

```
SELECT DenCl
FROM clienti_zile INNER JOIN zile ON NrZileCl = NrZile
```
[6]

Ce facturi conțin măcar produsele din factura 1117? (exemplul 27, paragraful 4.4.7)

```
WITH facturi_produse1117 AS (
                    SELECT lf1.NrFact, lf1.CodPr
                    FROM liniifact lf1 INNER JOIN liniifact lf2 ON
                         lf1.CodPr = lf2.CodPr AND lf2.NrFact = 1117
                    ),
      nrproduse_1117  AS (
               SELECT COUNT(DISTINCT CodPr) AS Nr
               FROM liniifact
               WHERE NrFact = 1117)
SELECT NrFact
FROM facturi_produse1117
GROUP BY NrFact
HAVING COUNT(DISTINCT CodPr) =
      (SELECT Nr
       FROM nrproduse_1117
      )
```

Aici am folosit în premieră o expresie-tabelă tocmai în subconsultarea din clauza HAVING, ceea ce nu era chiar evident. În general, orice interogare care ni s-a părut prea lungă/laborioasă poate fi simplicată cu ajutorul expresiilor-tabelă.

Care este contribuția (procentuală) a fiecărui produs la totalul vânzărilor?

```
WITH vanzari_produse AS
(
               SELECT DenPr AS Produs,
               SUM(Cantitate * PretUnit * (1 + ProcTVA))  AS Vanzari
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DenPr
),
total_vanzari AS
(
      SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Total
      FROM liniifact lf
               INNER JOIN produse p ON lf.CodPr = p.CodPr
)
SELECT Produs, Vanzari, Total,
```

6. Nu uitați să eliminați cuvintele DATE în variantele DB2 și SQL Server.

```
        CAST (Vanzari * 100/Total AS NUMERIC (4,2)) AS Procent
FROM vanzari_produse, total_vanzari
```

Pot apărea uşoare diferenţe faţă de figura 9.45, datorită folosirii funcţiei CAST în loc de TRUNC (din raţiuni de portabilitate).

Care sunt clienţii cu valoarea vânzărilor peste medie?

```
WITH
vanzari_clienti AS
        (SELECT DenCl AS client,
                ROUND(SUM(Cantitate *PretUnit *(1 + ProcTVA)),0) AS vanzari
        FROM clienti c
                INNER JOIN facturi f ON c.CodCl = f.CodCl
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DenCl),
vanzari_medii AS
        (SELECT SUM(Cantitate *PretUnit * (1 + ProcTVA))/
                COUNT(DISTINCT f.CodCl) AS medie
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr   )
SELECT *
FROM vanzari_clienti
WHERE vanzari > (SELECT medie FROM vanzari_medii)
```

Care este restul de plată al fiecărui client?

```
WITH
        facturat AS
                (SELECT DenCl, c.CodCl, COALESCE(SUM(Cantitate *
                                PretUnit * (1 + ProcTVA)),0) AS Facturat
                FROM clienti c
                        INNER JOIN facturi f ON c.CodCl = f.CodCl
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY DenCl, c.CodCl),
        incasat AS
                (SELECT CodCl, SUM(Transa) AS Incasat
                FROM facturi f INNER JOIN incasfact i ON f.NrFact = i.NrFact
                GROUP BY CodCl)
SELECT DenCl, Facturat, COALESCE(Incasat,0) AS Incasat,
```

```
        Facturat - COALESCE(Incasat,0) AS De_Incasat
FROM facturat f LEFT OUTER JOIN incasat i ON f.CodCl = i.CodCl
ORDER BY DenCl
```

Care este clientul cu cel mai mare rest de plată?

Bine mai zice proverbul cu „dai un deget şi ţi se ia toată mână". Forţând cu neruşinare SQL-ul, încercând încadrarea unei clauze WITH în altă clauză WITH:

```
WITH sinteza AS
 (WITH facturat AS (SELECT DenCl, c.CodCl, COALESCE(SUM(Cantitate *
                PretUnit * (1 + ProcTVA)),0) AS Facturat
                     FROM clienti c INNER JOIN facturi f ON c.CodCl = f.CodCl
                          INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                          INNER JOIN produse p ON lf.CodPr = p.CodPr
                     GROUP BY DenCl, c.CodCl),
      incasat AS (SELECT CodCl, SUM(Transa) AS Incasat
                FROM facturi f INNER JOIN incasfact i ON f.NrFact = i.NrFact
                GROUP BY CodCl)
      SELECT DenCl, Facturat, COALESCE(Incasat,0) AS Incasat,
                Facturat - COALESCE(Incasat,0) AS De_Incasat
      FROM facturat f LEFT OUTER JOIN incasat i ON f.CodCl = i.CodCl
      ORDER BY DenCl)
SELECT *
FROM sinteza
WHERE De_Incasat = (
                     SELECT MAX(De_Incasat) FROM sinteza
                     )
```

Reacţia Oracle la execuţia acestei interogări este suprinsă în figura 9.48. Nici DB2 sau SQL Server nu sunt mai cooperante. Nu putem bănui prea multă iritare în mesajul de eroare, deşi am fi meritat un pic de săpuneală. Reţinem, deci, că nu putem include o expresie-tabelă în alta.

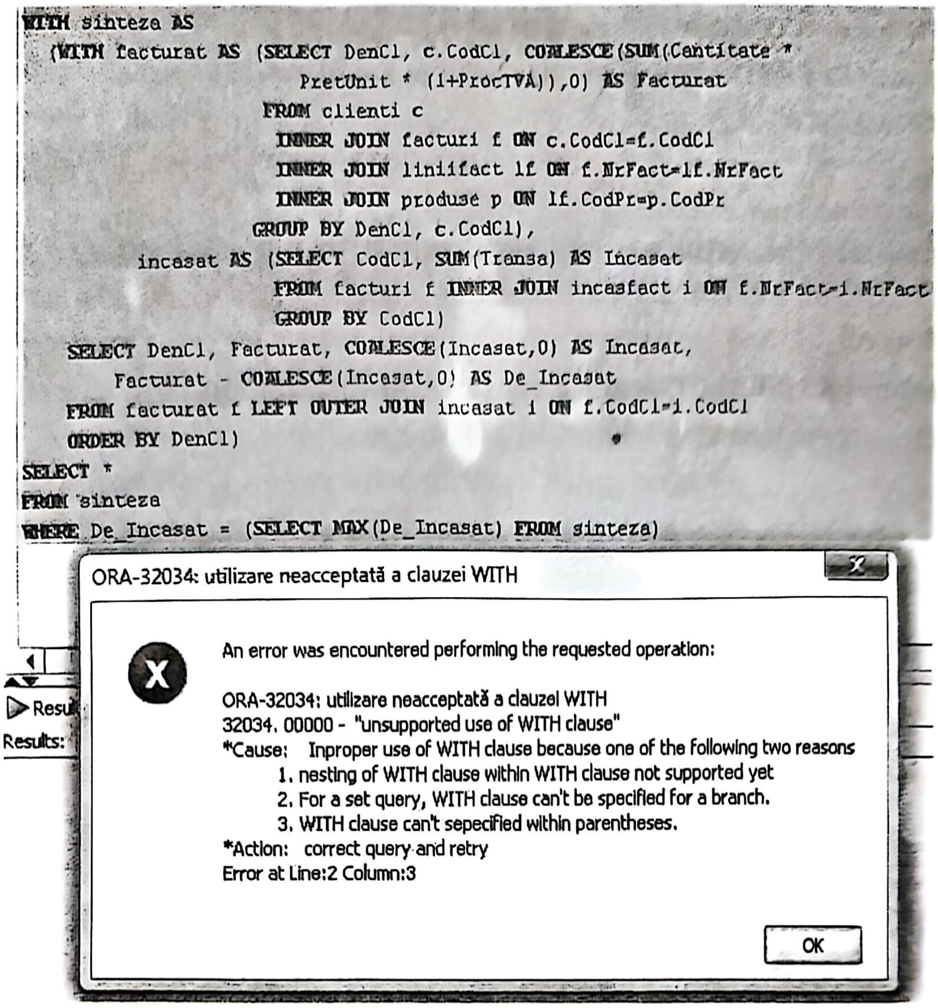


Figura 9.48. Expresiile tabelă nu se pot include una-n alta

Cu toate acestea, soluțiile bazate pe expresii-tabelă pentru această problemă nu lipsesc:

```
WITH situatie_clienti AS (
        SELECT CodCl,
                COALESCE(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Facturat,
                SUM(COALESCE(Transa,0)) AS Incasat
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
                LEFT OUTER JOIN incasfact i ON f.NrFact = i.NrFact
        GROUP BY CodCl)
SELECT DenCl, Facturat, Incasat, Facturat-Incasat AS RestPlata
FROM clienti c LEFT OUTER JOIN situatie_clienti sc ON c.CodCl = sc.CodCl
WHERE Facturat-Incasat =
                (SELECT MAX(Facturat-Incasat) FROM situatie_clienti)
```

sau:

```
WITH facturat AS
```

```
        (SELECT DenCl, c.CodCl, COALESCE(SUM(Cantitate *
                PretUnit * (1 + ProcTVA)),0) AS Facturat
        FROM clienti c
                INNER JOIN facturi f ON c.CodCl = f.CodCl
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DenCl, c.CodCl),
    incasat AS
        (SELECT CodCl, SUM(Transa) AS Incasat
        FROM facturi f INNER JOIN incasfact i ON f.NrFact = i.NrFact
        GROUP BY CodCl)
SELECT DenCl, Facturat, COALESCE(Incasat,0) AS Incasat,
    Facturat - COALESCE(Incasat,0) AS De_Incasat
FROM facturat f LEFT OUTER JOIN incasat i ON f.CodCl = i.CodCl
WHERE Facturat - COALESCE(Incasat,0) =
    (SELECT MAX(Facturat - COALESCE(Incasat,0))
    FROM facturat f LEFT OUTER JOIN incasat i ON f.CodCl = i.CodCl)
```

Care sunt numerele de facturi nefolosite?

Numerele facturilor emise sunt, de obicei, consecutive, datorită regimului acestui tip de documente. Totuşi, în tabela FACTURI avem câteva „găuri" (gap-uri, în romgleză) pe care ne propunem să le aflăm acum.

O soluţie la această problemă este de a genera ad-hoc o tabelă în care un rând să fie o cifră de la 0 la 9. Vom aplica produsul cartezian asupra a patru instanţe ale acestei tabele ah-hoc ştiind că, spre exemplu, 1123 = 1 * 1000 + 1 * 100 + 2 * 10 + 3:

```
WITH cifre AS (
    SELECT 0 AS Cifra FROM dual UNION SELECT 1 FROM dual
        UNION SELECT 2 FROM dual UNION SELECT 3 FROM dual
    UNION SELECT 4 FROM dual UNION
        SELECT 5 FROM dual UNION SELECT 6 FROM dual
    UNION SELECT 7 FROM dual UNION SELECT 8 FROM dual
    UNION SELECT 9 FROM dual
        )
SELECT c1000.Cifra * 1000 + c100.Cifra * 100 + c10.Cifra * 10 + c1.Cifra AS Numar
FROM cifre c1000
    CROSS JOIN cifre c100
    CROSS JOIN cifre c10
    CROSS JOIN cifre c1
ORDER BY 1 DESC
```

Rezutatul acestei interogări (vezi primele 7,25 linii din stânga figura 9.49) conţine 10000 de linii, ordonate de la 9999 la 0.

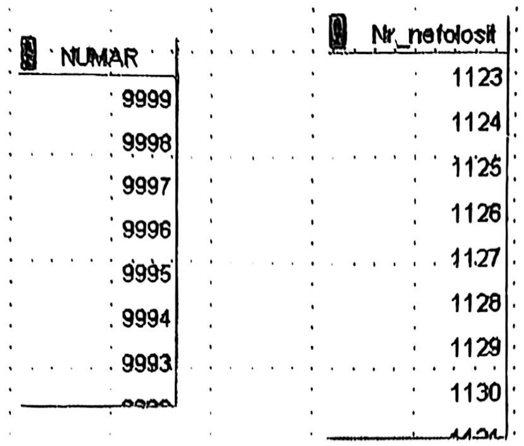


Figura 9.49. Numerele întregi descrescătoare de la 9999 la 0 (stânga) şi numere nefolosite în tabela FACTURI (dreapta)

Acum nu ne mai rămâne decât să folosim această tabelă drept argument al unui predicat în care vom căuta valorile care nu se regăsesc între numerele de factură minime şi maxime din tabela FACTURI (vezi dreapta figurii 9.49):

```
WITH cifre AS (
        SELECT 0 AS Cifra FROM dual UNION SELECT 1 FROM dual UNION
                SELECT 2 FROM dual UNION SELECT 3 FROM dual UNION
                        SELECT 4 FROM dual UNION
                SELECT 5 FROM dual UNION SELECT 6 FROM dual UNION
        SELECT 7 FROM dual UNION SELECT 8 FROM dual UNION
        SELECT 9 FROM dual
                )
SELECT Numar AS "Nr_nefolosit"
FROM (SELECT c1000.Cifra * 1000 + c100.Cifra * 100 + c10.Cifra * 10 + c1.Cifra
                AS Numar
        FROM cifre c1000 CROSS JOIN cifre c100 CROSS JOIN cifre c10
        CROSS JOIN cifre c1)
 WHERE Numar BETWEEN (SELECT MIN(NrFact) FROM facturi) AND
        (SELECT MAX(NrFact) FROM facturi)
                AND Numar NOT IN (SELECT NrFact FROM facturi)
 ORDER BY 1
```

Sintaxa interogării pentru celelalte trei servere BD rămâne ca temă pentru acasă.

Capitolul 10
Subconsultări corelate

Toate subconsultările din capitolul anterior aveau regim de autonomie, ca să nu-i zicem izolare. Clauzele SELECT, WHERE şi HAVING făceau apel doar la tabele enumerate în clauza FROM sau la expresii-tabelă (clauza WITH). Vom vedea, în continuare, că putem referi, într-o subconsultare, atribute ale unor tabele din consultarea principală (sau subconsultări superioare), un mare avantaj care însă ne va da şi destule bătăi de cap.

Sunt unul dintre cei care cred că piatra de încercare a unui SQL-ist o constituie subconsultările corelate. Nu atât din lipsa alternativelor, întrucât mai toate rezolvările din acest capitol au soluţii echivalente formulate sau sugerate în capitolele anterioare. Cel mai deconcertant aspect al interogărilor corelate este logica execuţiei acestora, care este una linie cu linie, nu ansamblistă, aşa cum am fost obişnuiţi până acum. Maximul de adrenalină este rezervat subconsultărilor dublu--corelate tratate în ultimul paragraf.

10.1. Subconsultări corelate în clauza SELECT

Am făcut cunoştinţă cu interogările scalare incluse în clauza SELECT în paragraful 9.5. Acum onorăm promisiunea făcută în finalul acelui paragraf, discutând despre subconsultări scalare incluse în clauza SELECT, care conţin referinţe către tabelele din clauza FROM a interogării principale.

Revenim, mai întâi, la o problemă simplă din paragraful 4.4.3, exemplul 9 (vezi figura 4.16) - *Care sunt codurile produselor care apar simultan în factura 1111 şi în factura 1117?* Alăturăm soluţiilor bazate pe intersecţie, joncţiune şi subconsultări (necorelate), una care nu este pe deplin satisfăcătoare, însă interesantă ca idee:

```
SELECT lf1.CodPr,
        (SELECT CodPr FROM liniifact lf2
        WHERE lf2.CodPr = lf1.CodPr AND lf2.NrFact = 1117) AS CodPr2
FROM liniifact lf1
WHERE NrFact = 1111
```

Prin clauzele FROM şi WHERE ale interogării principale vor fi selectate liniile facturii 1111 sub aliasul LF1. Prima coloană din rezultat conţine codurile celor trei produse 1, 2 şi 5 (vezi figura 10.1). Pentru fiecare dintre cele trei valori (pentru fiecare linie a rezultatului) se execută subconsultarea care se corelează cu fraza SELECT principală prin condiţia *lf2.CodPr = lf1.CodPr*. Practic, pentru fiecare linie a rezultatului ce corespunde codurilor produselor 1, 2 şi 5, subconsultările sunt:

- **SELECT CodPr FROM liniifact lf2 WHERE lf2.CodPr = 1 AND lf2.NrFact = 1117**
- **SELECT CodPr FROM liniifact lf2 WHERE lf2.CodPr = 2 AND lf2.NrFact = 1117**
- **SELECT CodPr FROM liniifact lf2 WHERE lf2.CodPr = 5 AND lf2.NrFact = 1117**

Prima „returnează" (deşi nu e funcţie) 1, a doua 2, iar a treia nimic, adică NULL.

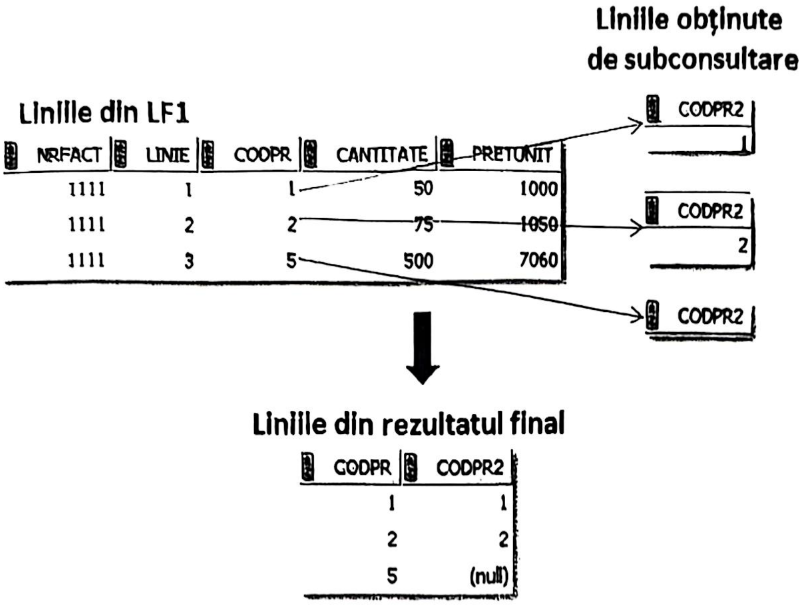


Figura 10.1. O subconsultare corelată în clauza SELECT

Rezervele legate de interogare sunt cauzate de faptul că rezultatul conţine două coloane, în prima fiind codurile produselor care apar în factura 1111, iar în a doua, aceleaşi coduri, dacă produsele respective apar şi în factura 1117 (produsele 1 şi 2), sau NULL, în caz contrar (produsul 5). Prin urmare, pentru a răspunde punctual la întrebare, trebuie să mai facem o filtrare, eliminând produsele la care, pe coloana a doua, apare NULL. Filtrarea este pur vizuală întrucât, după cum am văzut în paragraful 9.5, coloana definită prin subconsultarea scalară în clauza SELECT nu poate fi referită în clauza WHERE, deci (vorba lui Cristoiu!) interogarea *SELECT lf1.CodPr, (SELECT CodPr FROM liniifact lf2 WHERE lf2.CodPr = lf1.CodPr AND lf2.NrFact = 1117) AS CodPr2 FROM liniifact lf1 WHERE NrFact = 1111 AND CodPr2 IS NOT NULL* nu funcţionează. Pentru a nu înregistra un nou eşec, includem interogarea funcţională de mai sus (ce conţine subconsultarea scalară) în clauza FROM a unei interogări în care putem face filtrarea, rezultatul fiind cel din figura 10.2:

```
SELECT CodPr
FROM (SELECT lf1.CodPr, (
                    SELECT CodPr
                    FROM liniifact lf2
                    WHERE lf2.CodPr = lf1.CodPr AND lf2.NrFact = 1117
                        ) AS CodPr2
      FROM liniifact lf1
      WHERE NrFact = 1111) t
WHERE CodPr2 IS NOT NULL
```

CODPR
1
2

Figura 10.2. Codurile produselor ce apar şi în factura 1111, şi în factura 1117

Prin comparaţie cu simplitatea rezultatului, complexitatea interogării poate părea, totuşi, exagerată. Pentru a recicla un pic codul, putem folosi această soluţie pentru un exemplu legat de diferenţă:

Care sunt codurile produselor care apar în factura 1111, dar nu apar în factura 1117?

Singura modificare a interogării anterioare este eliminarea lui NOT de pe ultima linie:

```
SELECT CodPr
FROM (SELECT lf1.CodPr, (SELECT CodPr FROM liniifact lf2
                          WHERE lf2.CodPr = lf1.CodPr
                          AND lf2.NrFact = 1117
                          ) AS CodPr2
      FROM liniifact lf1
      WHERE NrFact = 1111 ) t
WHERE CodPr2 IS NULL
```

Care sunt localităţile în care se află sediul fiecărui client?

Dacă ultimele două probleme erau legate de intersecţie şi diferenţă, aceasta este una apropiată de joncţiune. Important este că tabela din clauza FROM a interogării principale este „copil", astfel încât subconsultarea extrage întotdeauna o singură linie (şi coloană) (vezi figura 10.3).

```
SELECT DenCl,
       (SELECT Loc
        FROM coduri_postale
        WHERE CodPost = clienti.CodPost
       ) AS Loc
FROM clienti ORDER BY 1
```

DENCL	LOC
Client 1 SRL	Iasi
Client 2 SA	Iasi
Client 3 SRL	Vaslui
Client 4	Pascani
Client 5 SRL	Timisoara
Client 6 SA	Roman
Client 7 SRL	Timisoara

Figura 10.3. Localitatea fiecărui client, obţinută printr-o subconsultare corelată

Care este valoarea vânzărilor din fiecare produs pentru luna septembrie 2007?

Având experienţa interogărilor anterioare, presupunem că interogarea următoare nu va întâmpină nicio rezistenţă[1]:

```
SELECT DenPr AS Produs,
        ROUND(
                COALESCE (
                        (SELECT SUM(Cantitate * PretUnit * (1 + p1.ProcTVA))
                        FROM liniifact INNER JOIN facturi
                                ON liniifact.NrFact = facturi.NrFact
                        WHERE DataFact BETWEEN '2007-09-01' AND '2007-09-30'
                                AND CodPr = p1.CodPr
                        ), 0)
                ,0) AS Vanzari_Sept2007
FROM produse p1
ORDER BY DenPr
```

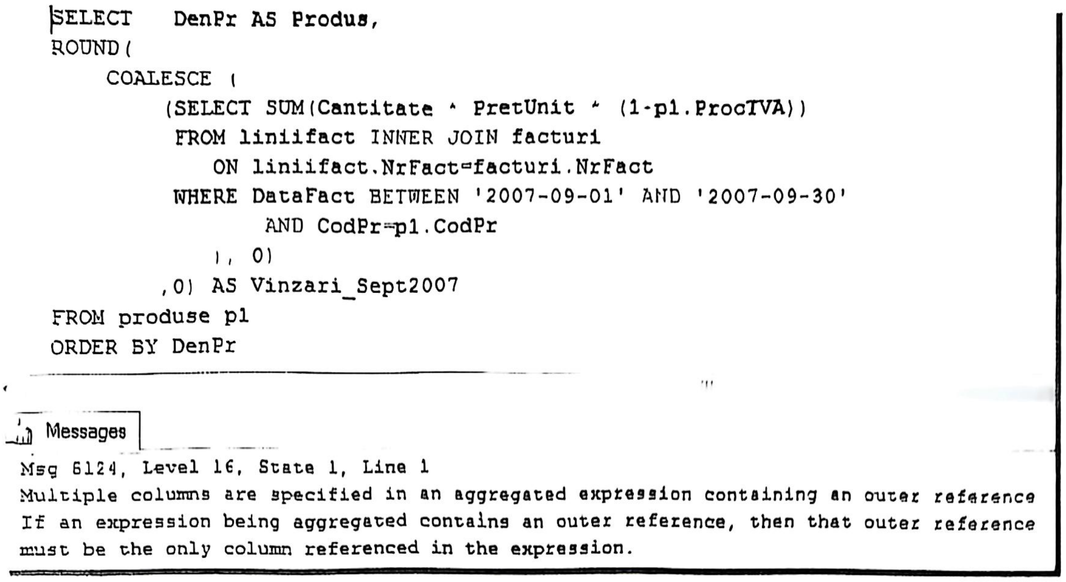
Aşa se şi întâmplă în PostgreSQL, Oracle (cu modificările legate de DATE) şi DB2 (vezi figura 10.4).

PRODUS	VINZARI_SEPT2007
Produs 1	324,989.0000
Produs 2	771,393.0000
Produs 3	58,013.0000
Produs 4	0.0000
Produs 5	4,284,714.0000
Produs 6	0.0000

Figura 10.4. Valoarea vânzărilor din septembrie 2007 pentru fiecare produs, obţinută printr-o subconsultare corelată (DB2)

SQL Server-ul ne face cadou însă o eroare destul de tulburătoare (vezi figura 10.5). Dacă ar fi să traducem mesajul, s-ar înţelege că în subconsultarea corelată singurul atribut extern (din tabele ale clauzei FROM din interogarea principală) ce poate fi folosit în expresia SUM (...) ar fi cel după care se face corelarea, adică p1.CodPr...

1. În Oracle trebuie adăugat cuvântul DATE înaintea constantelor de tip dată calendaristică.



```
SELECT   DenPr AS Produs,
ROUND(
    COALESCE (
        (SELECT SUM(Cantitate * PretUnit * (1-p1.ProcTVA))
         FROM liniifact INNER JOIN facturi
            ON liniifact.NrFact=facturi.NrFact
         WHERE DataFact BETWEEN '2007-09-01' AND '2007-09-30'
               AND CodPr=p1.CodPr
            ), 0)
        ,0) AS Vinzari_Sept2007
FROM produse p1
ORDER BY DenPr
```

Messages

```
Msg 6124, Level 16, State 1, Line 1
Multiple columns are specified in an aggregated expression containing an outer reference
If an expression being aggregated contains an outer reference, then that outer reference
must be the only column referenced in the expression.
```

Figura 10.5. Eroare SQL Server la folosirea interogărilor corelate

Care este contribuția (procentuală) a fiecărui produs la totalul vânzărilor?

Ne vom servi de o subconsultare scalară pentru a determina, pe fiecare linie (corespunzătoare unui produs), totalul vânzărilor și a calcula, astfel, raportul care ne interesează (vezi figura 10.6).

```
SELECT DenPr AS Produs,
        COALESCE(
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM liniifact lf
                WHERE lf.CodPr = produse.CodPr
                ), 0) AS Vanzari_Produs,
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
        FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
        ) AS Total_Vanzari,
        ROUND( COALESCE(
                        (
                (
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
        FROM liniifact lf
        WHERE lf.CodPr = produse.CodPr
        )
         * 100 )/(
                SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
                        )
                                ),0 ), 2 ) AS Procent
```

```
FROM produse
ORDER BY DenPr
```

PRODUS	VINZARI_PRODUS	TOTAL_VINZARI	PROCENT
Produs 1	988533	49170335	2,01
Produs 2	2784023,5	49170335	5,66
Produs 3	251685	49170335	0,51
Produs 4	301657,5	49170335	0,61
Produs 5	44844436	49170335	91,2
Produs 6	0	49170335	0

Figura 10.6. Contribuția fiecărui produs la totalul vânzărilor

Din nou SQL Server-ul se opune la acest gen de corelare, mesajul afișat fiind similar celui de la problema precedentă.

Să se afișeze numărul curent al fiecărei facturi emise în luna septembrie 2007.

Acest gen de problemă își are cea mai elegantă rezolvare prin folosirea funcției ROW_NUMBER(), una dintre funcțiile OLAP prezentate în capitolul următor. Până atunci, însă, putem formula o interogarea simplă bazată pe o subconsultare în clauza FROM. Soluția este cu atât mai importantă pentru PostgreSQL în condițiile în care în acest produs nu (prea) au fost implementate funcții OLAP[2] și nici opțiunea ROWNUM din Oracle. Ideea este următoare: prin clauzele FROM și WHERE sunt selectate numai facturile din sept. 2007. Pentru fiecare linie (factură din septembrie) se vor număra, printr-o subconsultare (scalară și corelată) din clauza SELECT câte facturi din septembrie au numărul mai mic decât cel al facturii curente (linia curentă din f2):

```
SELECT (
        SELECT COUNT(*) FROM facturi f2
        WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH
                        FROM DataFact) = 9 AND NrFact <= f1.NrFact
        ) AS NrCrt_Sept07, f1.*
FROM facturi f1
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
```

nrcrt_sept07 bigint	nrfact integer	datafact date	codcl smallint	obs character varying(50)
1	3111	2007-09-01	1001	
2	3112	2007-09-01	1005	Probleme cu transportul
3	3113	2007-09-02	1002	
4	3115	2007-09-02	1001	
5	3116	2007-09-10	1007	Pretul propus initial a fost modificat
6	3117	2007-09-10	1001	
7	3118	2007-09-17	1001	

Figura 10.7. Numărul curent al fiecărei facturi din luna septembrie 2007

2. Idee preluată din *http://www.slideshare.net/xzilla/pro-postgresql-389802/*.

Reuşita interogării ne este confirmată de figura 10.7. Înlocuind funcţiile EXTRACT cu YEAR şi MONTH obţinem varianta DB2/SQL Server.

Să se afişeze numărul curent al fiecărei facturi în cadrul lunii în care a fost emisă.

Diferenţa faţă de problema anterioară ţine de *resetarea* numărului curent al facturilor la începutul fiecărei luni calendaristice (vezi figura 10.8).

nrcrt_pt_lună bigint	nrfact integer	datafact date	codcl smallint	obs character varying(50)
1	1111	2007-08-01	1001	
2	1112	2007-08-01	1005	Probleme cu transportul
3	1113	2007-08-01	1002	
4	1114	2007-08-01	1006	
5	1115	2007-08-02	1001	
6	1116	2007-08-02	1007	Pretul propus initial a fost mo
7	1117	2007-08-03	1001	
8	1118	2007-08-04	1001	
9	1119	2007-08-07	1003	
10	1120	2007-08-07	1001	
11	1121	2007-08-07	1004	
12	1122	2007-08-07	1005	
13	2111	2007-08-14	1001	
14	2112	2007-08-14	1005	Probleme cu transportul
15	2113	2007-08-14	1002	
16	2115	2007-08-15	1001	
17	2116	2007-08-15	1007	Pretul propus initial a fost mo
18	2117	2007-08-16	1001	
19	2118	2007-08-16	1001	
20	2119	2007-08-21	1003	
21	2121	2007-08-21	1004	
22	2122	2007-08-22	1005	
1	3111	2007-09-01	1001	
2	3112	2007-09-01	1005	Probleme cu transportul
3	3113	2007-09-02	1002	

Figura 10.8. Numărul curent al fiecărei facturi în luna în care a fost emisă

Logica interogării este aceeaşi, doar că se schimbă expresia de corelare:

```
SELECT (
        SELECT COUNT(*) FROM facturi f2 WHERE EXTRACT (YEAR
                FROM f2.DataFact) = EXTRACT (YEAR FROM f1.DataFact)
                AND EXTRACT (MONTH FROM f2.DataFact) = EXTRACT (MONTH
                        FROM f1.DataFact) AND f2.NrFact <= f1.NrFact
        ) AS NrCrt_Pt_LunaCrt, f1.*
FROM facturi f1
```

Care este evoluţia zilnică a vânzărilor, în raport cu ziua calendaristică anterioară?

Ne interesează un rezultat precum cel din figura 10.9. Dificultatea principală ţine de sincronizarea dintre ziua curentă de vânzări şi ziua precedentă.

ZI	VINZARI_ZI_CURENTA	VINZARI_ZI_PRECEDENTA	DIFERENTA
01-08-2007	10599535	0	10599535
02-08-2007	277950	10599535	-10321585
03-08-2007	222050	277950	-55900
04-08-2007	201975	222050	-20075
07-08-2007	10610000,5	0	10610000,5
14-08-2007	4723931,5	0	4723931,5
15-08-2007	247757	4723931,5	-4476174,5
16-08-2007	467420	247757	219663
21-08-2007	10560939,5	0	10560939,5
01-09-2007	4596401,5	0	4596401,5
02-09-2007	238437,5	4596401,5	-4357964
10-09-2007	424704,5	0	424704,5
17-09-2007	179565	0	179565
07-10-2007	5819668	0	5819668

Figura 10.9. Vânzări zilnice, comparate cu cele din ziua precedentă

Iată soluţia Oracle:

```
SELECT DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi_Curenta,
        COALESCE (
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM facturi f2
        INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                        INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
                WHERE DataFact = f.DataFact - INTERVAL '1' DAY
                        ),0) AS Vanzari_Zi_Precedenta,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) -
                COALESCE(
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM facturi f2
        INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
                        WHERE DataFact = f.DataFact - INTERVAL '1' DAY
        ),0) AS Diferenta
FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DataFact
ORDER BY DataFact
```

Prima interogare scalară furnizează totalul vânzărilor pentru ziua precedentă, ziua curentă fiind cea a grupului de referinţă (din SELECT-ul principal). A doua

instanță a interogării scalare permite calcularea diferenței dintre vânzările zilei curente și a celei precedente. Funcția COALESCE convertește eventuala valoare NULL datorată inexistenței vânzărilor în data calendaristică precedentă (așa cum este cazul zilelor de 1, 7, 14 și 21 august 2007, 1, 10 și 17 septembrie și 7 octombrie) în zero.

Schimbând *INTERVAL '1' DAY* cu *1*, obținem varianta care funcționează în MS SQL Server, iar cu *1 DAY* o obținem pe cea DB2. Lansarea interogării (nemodificate) în PostgreSQL ne dă prilejul unei noi suprize neplăcute. Rezultatul furnizat este altul (vezi figura 10.10). Pentru fiecare linie a rezultatului, vânzările din ziua curentă sunt identice cu cele din ziua precedentă.

zi date	vinzari_zi_curenta numeric	vinzari_zi_precedenta numeric	diferenta numeric
2007-08-01	10599535.0000	10599535.0000	0.0000
2007-08-02	277950.0000	277950.0000	0.0000
2007-08-03	222050.0000	222050.0000	0.0000
2007-08-04	201975.0000	201975.0000	0.0000
2007-08-07	10610000.5000	10610000.5000	0.0000
2007-08-14	4723931.5000	4723931.5000	0.0000
2007-08-15	247757.0000	247757.0000	0.0000
2007-08-16	467420.0000	467420.0000	0.0000
2007-08-21	10560939.5000	10560939.5000	0.0000
2007-09-01	4596401.5000	4596401.5000	0.0000
2007-09-02	238437.5000	238437.5000	0.0000
2007-09-10	424704.5000	424704.5000	0.0000
2007-09-17	179565.0000	179565.0000	0.0000
2007-10-07	5819668.0000	5819668.0000	0.0000

Figura 10.10. Un pic de năuceală proaspătă în PostgreSQL

Neavând o explicație științifică a fenomenului oarecum paranormal, bănuim, totuși, că ni se trage de la modul de specificare a intervalului, prilej cu care rememorăm o discuție purtată în paragraful 5.3. Așa e, pentru că, dacă înlocuim *INTERVAL '1' DAY* cu *INTERVAL '1 DAYS'*:

```
SELECT DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi_Curenta,
        COALESCE       (
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM facturi f2
                        INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                        INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
                WHERE DataFact = F.DataFact - INTERVAL '1 DAYS'
                        ),0) AS Vanzari_Zi_Precedenta,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) -
                COALESCE(
```

```
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
          FROM facturi f2
                INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
          WHERE DataFact = F.DataFact - INTERVAL '1 DAYS'
                ),0) AS Diferenta
FROM facturi f INNER JOIN liniifact lf ON f.Nrfact = lf.NrFact
     INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DataFact ORDER BY DataFact
```

rezultatul va fi corect şi în PostgreSQL.

O limită a corelării în clauza SELECT iese la iveală atunci când încercăm să simplificăm interogarea de mai sus cu ajutorul unei subconsultări în clauza FROM. Încercăm ca, printr-o subconsultare în clauza FROM, să creăm o tabelă ad-hoc – VÂNZĂRI_ZILNICE – la care să ne corelăm în clauza SELECT:

```
SELECT Zi, Vanzari_Zi,
     COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
          WHERE vz2.Zi = vanzari_zilnice.Zi - INTERVAL '1' DAY),
                0) AS Vanzari_zi_precendenta,
     Vanzari_Zi - COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
          WHERE vz2.Zi = vanzari_zilnice.Zi - INTERVAL '1' DAY),0)
                AS Diferenta
FROM
     (SELECT DataFact AS Zi,
          SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi
     FROM facturi f
          INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
          INNER JOIN produse p ON lf.CodPr = p.CodPr
     GROUP BY DataFact
     ORDER BY DataFact
     ) vanzari_zilnice
```

Interogarea s-ar simplifica destul de mult, însă nici Oracle, nici DB2, nici PostgreSQL nu recunosc în subconsultarea din clauza FROM tabela „ad-hoc" (SQL Server-ul nici atât) - vezi figura 10.11.

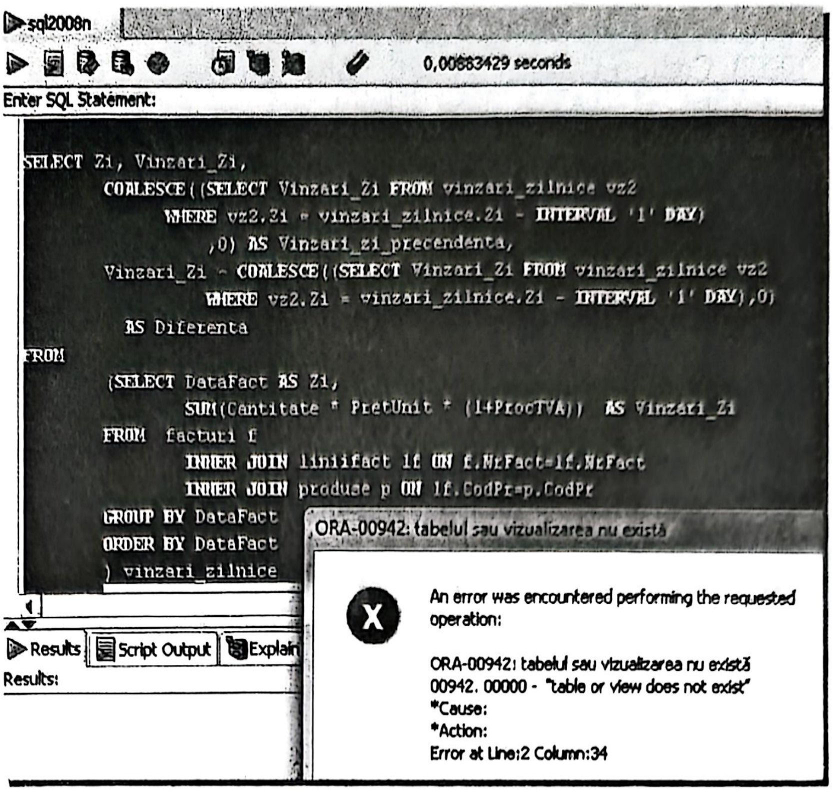
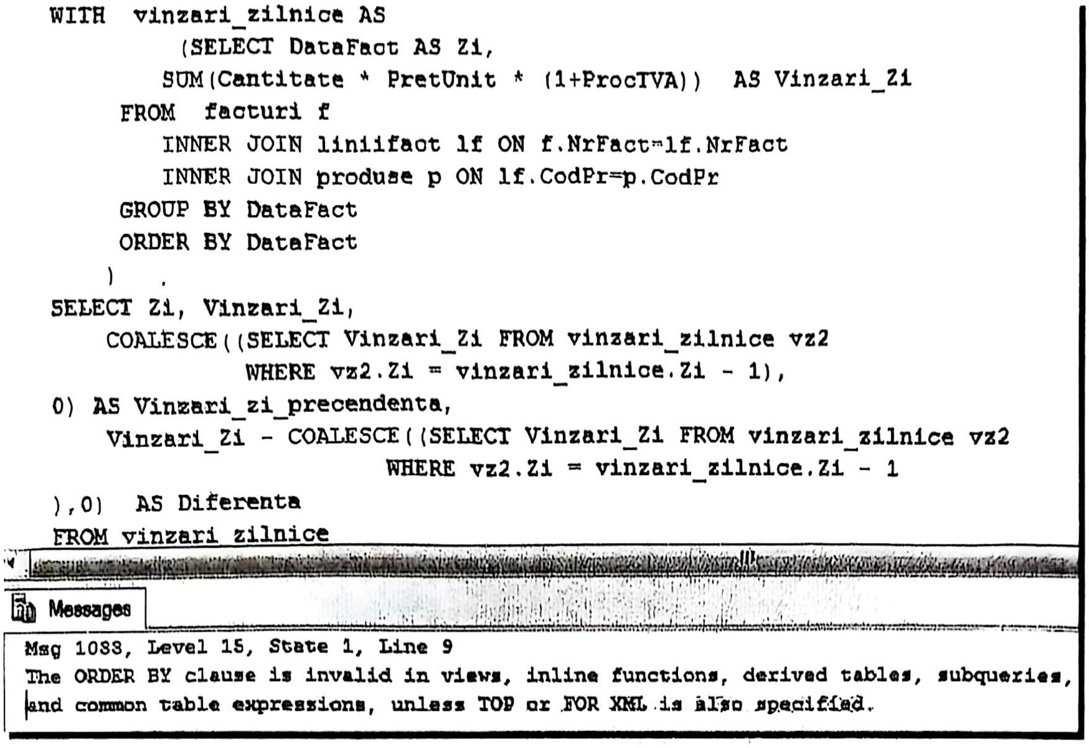


Figura 10.11. O subconsultare din clauza SELECT nu se poate corela (întotdeauna) cu o tabelă ad-hoc obținută printr-o subconsultarea în clauza FROM

Pentru a nu înregistra o (altă) înfrângere, vom transpune logica acestei interogări în alta similară (Oracle) ce foloseşte o expresie-tabelă.

```
WITH vanzari_zilnice AS
        (SELECT DataFact AS Zi,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DataFact
        ORDER BY DataFact
        )
SELECT Zi, Vanzari_Zi,
        COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
        WHERE vz2.Zi = vanzari_zilnice.Zi - INTERVAL '1' DAY),
                0) AS Vanzari_zi_precendenta,
        Vanzari_Zi - COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
                WHERE vz2.Zi = vanzari_zilnice.Zi - INTERVAL '1' DAY
                                ),0) AS Diferenta
 FROM vanzari_zilnice
```

Portarea acesteia în SQL Server ne dă prilejul descoperirii unei noi probleme (vezi figura 10.12). Conform mesajului, ordonarea din expresia-tabelă este invalidă în lipsa opţiunii TOP.



```
WITH  vinzari_zilnice AS
        (SELECT DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1+ProcTVA))  AS Vinzari_Zi
     FROM  facturi f
        INNER JOIN liniifact lf ON f.NrFact=lf.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
     GROUP BY DataFact
     ORDER BY DataFact
     )  .
SELECT Zi, Vinzari_Zi,
    COALESCE((SELECT Vinzari_Zi FROM vinzari_zilnice vz2
              WHERE vz2.Zi = vinzari_zilnice.Zi - 1),
0) AS Vinzari_zi_precendenta,
    Vinzari_Zi - COALESCE((SELECT Vinzari_Zi FROM vinzari_zilnice vz2
                         WHERE vz2.Zi = vinzari_zilnice.Zi - 1
),0)  AS Diferenta
FROM vinzari zilnice

Messages
Msg 1033, Level 15, State 1, Line 9
The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries,
and common table expressions, unless TOP or FOR XML is also specified.
```

Figura 10.12. O problemă SQL Server de ordonare într-o expresie-tabelă

Ei bine, top a vrut, top a găsit! Includem, din propria burtă, un *TOP 1000* pentru a salva sintactic interogarea:

```
WITH vanzari_zilnice AS
    (SELECT TOP 1000 DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Zi
    FROM facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
    GROUP BY DataFact
    ORDER BY DataFact
    )
SELECT Zi, Vanzari_Zi,
    COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
              WHERE vz2.Zi = vanzari_zilnice.Zi - 1),
              0) AS Vanzari_zi_precendenta,
    Vanzari_Zi - COALESCE((SELECT Vanzari_Zi FROM vanzari_zilnice vz2
              WHERE vz2.Zi = vanzari_zilnice.Zi - 1
              ),0) AS Diferenta
FROM vanzari_zilnice
```

Acum merge!

Să se afişeze penalizările pentru întârzierea la plată a facturilor, ştiind că acestea se calculează pentru fiecare zi de întârziere, pe tranşe, astfel:

- pentru tranşele neîncasate în maxim zece zile de la data facturării, nu se calculează penalizări;
- pentru tranşele neîncasate în intervalul 11-12 zile de la data facturării, procentul de penalitate este 0,05% din valoarea neîncasată, pentru fiecare zi de întârziere;
- pentru tranşele neîncasate în intervalul 13-14 zile de la data facturării, procentul de penalitate este 0,1% din valoarea neîncasată, pentru fiecare zi de întârziere;
- pentru tranşele încasate în intervalul 15-20 zile de la data facturării, procentul este de 0,2% din valoarea neîncasată pentru fiecare zi de întârziere;
- pentru tranşele încasate în intervalul 21-40 zile de la data facturării, procentul este de 0,3% din valoarea neîncasată pentru fiecare zi de întârziere;
- pentru tranşele încasate cu întârziere de peste 40 zile de la data facturării, procentul este de 0,5% din valoarea neîncasată pentru fiecare zi de întârziere;

Iniţial, această problemă a fost formulată în finalul paragrafului 9.4.1, moment în care am recunoscut că suntem incapabili să o rezolvăm. Pe calapodul soluţiilor anterioare, ne folosim de o subconsultare corelată pentru a determina încasările cumulate, până în momentul unei tranşe, şi, astfel, să aflăm restul de încasat:

```
SELECT NrFact AS "NrFact", DataFact AS "DataFact", Facturat AS "Facturat",
        DataInc AS "DataIncas", Incasat AS "Incasat", Zile_Intarz AS "Zile_Intarziere",
        Proc_Penaliz AS "Proc_Penalz",
                COALESCE((SELECT SUM(Incasat) FROM incasari i2
                        WHERE NrFact = t.NrFact AND DataInc < t.DataInc),0)
                                AS "Incas_Precedente"
FROM (SELECT vanzari.NrFact, DataFact, Facturat,
                COALESCE(DataInc, CURRENT_DATE) AS DataInc,
                COALESCE(Incasat,0) AS Incasat,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        AS NrZile,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact - 10,0)
                        AS Zile_Intarz,
                CASE
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 10 THEN 0
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 12
                        THEN 0.0005
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 14
                        THEN 0.001
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 20
                        THEN 0.002
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 40
                        THEN 0.003
```

```
               ELSE 0.005
               END AS Proc_Penaliz
FROM     (SELECT f.NrFact, DataFact, ROUND(SUM(Cantitate * PretUnit
                    * (1 + ProcTVA)),0) AS Facturat
               FROM facturi f
                    INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
               GROUP BY f.NrFact, DataFact
               ) VANZARI
                              LEFT OUTER JOIN
               (SELECT NrFact, DataInc, ROUND(SUM(Transa),0) AS Incasat
               FROM incasfact INNER JOIN incasari
                    ON incasfact.CodInc = incasari.CodInc
               GROUP BY NrFact, DataInc
               ) INCASARI
                              ON VANZARI.NrFact = INCASARI.NrFact
ORDER BY NrFact, DataInc) t
```

Dacă ar fi fost să ne luăm după figura 10.11, interogarea corelată menită să calculeze încasările tranşei precedente (pentru factura respectivă) - interogare subliniată prin caracterele italice - ar fi trebuit să nu funcţioneze. Ei bine, avem o veste bună şi una rea. Cea bună este că funcţionează în Oracle (vezi figura 10.13). Cea proastă este că funcţionează aiurea (vezi tot figura 10.13).

NrFact	DataFact	Facturat	DataIncas	Incasat	Zile_Intirziere	Proc_Penalz	Incas_Precedente
1111	01-08-2007	4346038	15-08-2007	53996	4	0,001	0
1112	01-08-2007	125516	15-08-2007	125516	4	0,001	0
1113	01-08-2007	106275	17-08-2007	106275	6	0,002	318825
1114	01-08-2007	6021707	12-03-2008	0	214	0,005	0
1115	02-08-2007	151238	12-03-2008	0	213	0,005	0
1116	02-08-2007	126713	12-03-2008	0	213	0,005	0
1117	03-08-2007	222050	16-08-2007	9754	3	0,001	19508
1117	03-08-2007	222050	17-08-2007	9754	4	0,001	29262
1117	03-08-2007	222050	18-08-2007	3696	5	0,002	18480
1118	04-08-2007	201975	15-08-2007	101975	1	0,0005	0
1118	04-08-2007	201975	16-08-2007	100000	2	0,0005	200000
1119	07-08-2007	5774499	12-03-2008	0	208	0,005	0
1120	07-08-2007	97664	16-08-2007	7315	-1	0	14630
1121	07-08-2007	4737838	12-03-2008	0	208	0,005	0

Figura 10.13. O eroare (SQL) de corelare

Dacă la facturile 1111 şi 1112 încasările precedente sunt corecte, pentru factura 1113 valoarea Încas_Precente este cel puţin curioasă. Ca să nu mai vorbim de situaţia facturii 1117. Ia să înlocuim interogarea corelată din clauza SELECT (cea în *italice* în interogarea de mai sus) cu una în care calculăm încasările precedente din tabelele ÎNCASĂRI şi ÎNCASFACT:

```
SELECT NrFact AS "NrFact", DataFact AS "DataFact", Facturat AS "Facturat", DataInc
        AS "DataIncas", Incasat AS "Incasat", Zile_Intarz AS "Zile_Intarziere",
        Proc_Penaliz AS "Proc_Penalz", COALESCE((SELECT SUM(Transa)
                FROM incasari i2 INNER JOIN incasfact if2 ON i2.CodInc = if2.CodInc
                        WHERE NrFact = t.NrFact AND DataInc < t.DataInc)
                                ,0) AS "Incas_Precedente"
FROM
        (SELECT vanzari.NrFact, DataFact, Facturat,
                COALESCE(DataInc, CURRENT_DATE) AS DataInc,
                COALESCE(Incasat,0) AS Incasat,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        AS NrZile,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact - 10,0)
                        AS Zile_Intarz,
                CASE
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 10 THEN 0
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 12
                        THEN 0.0005
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 14
                        THEN 0.001
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 20
                        THEN 0.002
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 40
                        THEN 0.003
                ELSE 0.005
                END AS Proc_Penaliz
        FROM
                (SELECT f.NrFact, DataFact, ROUND(SUM(Cantitate * PretUnit
                        * (1 + ProcTVA)),0) AS Facturat
                FROM facturi f
                        INNER JOIN liniifact lf ON f.NrFact=lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY f.NrFact, DataFact
                ) VANZARI
                                LEFT OUTER JOIN
                (SELECT NrFact, DataInc, ROUND(SUM(Transa),0) AS Incasat
                FROM incasfact INNER JOIN incasari
                        ON incasfact.CodInc = incasari.CodInc
                GROUP BY NrFact, DataInc
                ) INCASARI
```

```
                    ON VANZARI.NrFact = INCASARI.NrFact
    ORDER BY NrFact, DataInc
    ) t
```

Rezultatul este cât se poate de corect - vezi figura 10.14 - valorile încasărilor precedente fiind calculate cum se cuvine pentru facturile 1117 şi 1118, iar pentru factura 1120 numărul zilelor de întârziere este negativ (încasarea a fost mai devreme cu o zi.

NrFact	DataFact	Facturat	DataIncas	Incasat	Zile_Intirziere	Proc_Penalz	Incas_Precedente
1111	01-08-2007	4346038	15-08-2007	53996	4	0,001	0
1112	01-08-2007	125516	15-08-2007	125516	4	0,001	0
1113	01-08-2007	106275	17-08-2007	106275	6	0,002	0
1114	01-08-2007	6021707	12-03-2008	0	214	0,005	0
1115	02-08-2007	151238	12-03-2008	0	213	0,005	0
1116	02-08-2007	126713	12-03-2008	0	213	0,005	0
1117	03-08-2007	222050	16-08-2007	9754	3	0,001	0
1117	03-08-2007	222050	17-08-2007	9754	4	0,001	9754
1117	03-08-2007	222050	18-08-2007	3696	5	0,002	19508
1118	04-08-2007	201975	15-08-2007	101975	1	0,0005	0
1118	04-08-2007	201975	16-08-2007	100000	2	0,0005	101975
1119	07-08-2007	5774499	12-03-2008	0	208	0,005	0
1120	07-08-2007	97664	16-08-2007	7315	-1	0	0
1121	07-08-2007	4737838	12-03-2008	0	208	0,005	0

Figura 10.14. Modificarea (cu succes a) subconsultării corelate

Mai avem, însă, de lucru până la furnizarea răspunsului direct la întrebarea formulată. Mai întâi, pentru a afla întârzierea pentru fiecare tranşă, ar trebui să înmulţim - repet, pentru fiecare tranşă - procentul aferent numărului de zile de întârziere a tranşei cu diferenţa dintre valoarea totală a facturii şi încasările precedente tranşei din factură:

```
SELECT NrFact AS "NrFact", DataFact AS "DataFact", Facturat AS "Facturat", DataInc
    AS "DataIncas", Incasat AS "Incasat", Zile_Intarz AS "Zile_Intarziere",
    Proc_Penaliz AS "Proc_Penalz", COALESCE((SELECT SUM(Transa)
            FROM incasari i2 INNER JOIN incasfact if2 ON i2.CodInc = if2.CodInc
                WHERE NrFact = t.NrFact AND DataInc < t.DataInc)
                    ,0) AS "Incas_Precedente",
    Proc_Penaliz * (Facturat - COALESCE((SELECT SUM(Transa) FROM
            incasari i2 INNER JOIN incasfact if2 ON i2.CodInc = if2.CodInc
            WHERE NrFact = t.NrFact AND DataInc < t.DataInc)
                    , 0)) AS "Penaliz_Transa"
FROM
    (...) t
```

Ceea ce obținem (figura 10.15) este încă un pas către rezultatul final. Penalizările fiecărei tranșe par a fi corecte, dar sunt cele din momentul încasării tranșei respective. Dacă pentru facturile fără nicio încasare penalizarea se calculează corect (relativ la momentul curent[3]), mai există însă o serie de penalizări necalculate pentru sumele rămase de încasat din fiecare factură plătită parțial.

NrFact	DataFact	Facturat	DataIncas	Incasat_Transa	Zile_Intirziere	Proc_Penalz	Incas_Precedente	Penaliz_Transa
1111	01-08-2007	4346038	15-08-2007	53996	4	0,001	0	4346,038
1112	01-08-2007	125516	15-08-2007	125516	4	0,001	0	125,516
1113	01-08-2007	106275	17-08-2007	106275	6	0,002	0	212,55
1114	01-08-2007	6021707	12-03-2008	0	214	0,005	0	30108,535
1115	02-08-2007	151238	12-03-2008	0	213	0,005	0	756,19
1116	02-08-2007	126713	12-03-2008	0	213	0,005	0	633,565
1117	03-08-2007	222050	16-08-2007	9754	3	0,001	0	222,05
1117	03-08-2007	222050	17-08-2007	9754	4	0,001	9754	212,296
1117	03-08-2007	222050	18-08-2007	3696	5	0,002	19508	405,084
1118	04-08-2007	201975	15-08-2007	101975	1	0,0005	0	100,9875
1118	04-08-2007	201975	16-08-2007	100000	2	0,0005	101975	50
1119	07-08-2007	5774499	12-03-2008	0	208	0,005	0	28872.495
1120	07-08-2007	97664	16-08-2007	7315	-1	0	0	0
1121	07-08-2007	4737838	12-03-2008	0	208	0,005	0	23689,19

Figura 10.15. Modificarea (cu succes a) subconsultării corelate

Cum singura problemă rămasă de rezolvat este includerea în rezultat a facturilor încasate parțial la data curentă (12 martie, în cazul meu), apelăm la un truc ieftin: includem în tabela ad-hoc ÎNCASĂRI2 câte o linie pentru fiecare factură încasată parțial în care data încasării să fie data curentă, iar tranșa încasată să fie zero:

```
SELECT NrFact AS "NrFact", DataFact AS "DataFact", Facturat AS "Facturat",
        DataInc AS "DataIncas", Incasat AS "Incasat_Transa",
        Zile_Intarz AS "Zile_Intarziere", Proc_Penaliz AS "Proc_Penalz",
        COALESCE((SELECT SUM(Transa)
                FROM incasari i2 INNER JOIN incasfact if2
                        ON i2.CodInc = if2.CodInc
                WHERE NrFact = t.NrFact AND DataInc < t.DataInc),0
                ) AS "Incas_Precedente",
        Proc_Penaliz * (Facturat - COALESCE(
                (SELECT SUM(Transa)
                FROM incasari i2
                        INNER JOIN incasfact if2 ON i2.CodInc = if2.CodInc
                WHERE NrFact = t.NrFact AND DataInc < t.DataInc),0))
                        AS "Penaliz_Transa"
```

3. Interogările pentru rezolvarea acestei probleme au fost executate pe 12 martie 2008 (vezi unde apare CURRENT_DATE).

```
FROM
        (SELECT vanzari.NrFact, DataFact, Facturat,
                COALESCE(DataInc, CURRENT_DATE) AS DataInc,
                COALESCE(Incasat,0) AS Incasat,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact,0)
                        AS NrZile,
                TRUNC(COALESCE(DataInc, CURRENT_DATE) - DataFact - 10,0)
                        AS Zile_Intarz,
                CASE
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 10
                        THEN 0
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 12
                        THEN 0.0005
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 14
                        THEN 0.001
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 20
                        THEN 0.002
                WHEN COALESCE(DataInc,CURRENT_DATE) - DataFact <= 40
                        THEN 0.003
                ELSE 0.005
                END AS Proc_Penaliz
        FROM
                (SELECT f.NrFact, DataFact, ROUND(SUM(Cantitate *
                        PretUnit * (1 + ProcTVA)),0) AS Facturat
                FROM facturi f
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY f.NrFact, DataFact
                ) VANZARI
                                LEFT OUTER JOIN
                (SELECT NrFact, DataInc, ROUND(SUM(Transa),0) AS Incasat
                FROM incasfact INNER JOIN incasari
                        ON incasfact.CodInc = incasari.CodInc
                 GROUP BY NrFact, DataInc
                        UNION
                        SELECT NrFact, CURRENT_DATE, 0
                        FROM incasfact i
                        GROUP BY NrFact
                        HAVING SUM(Transa) < (
```

```
                SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM liniifact lf INNER JOIN produse p
                        ON lf.CodPr = p.CodPr
                WHERE NrFact = i.Nrfact
                        )
        ) INCASARI2
                ON VANZARI.NrFact = INCASARI2.NrFact
    ORDER BY NrFact, DataInc
    ) t
```

Figura 10.16 este încurajatoare, întrucât apare câte o linie nouă pentru fiecare factură din care s-a încasat câte ceva, dar mai rămâne un rest de încasat cât de mic - facturile 1111 şi 1117 (probabil mai sunt şi altele, dar n-au încăput în figură).

NrFact	DataFact	Facturat	DataIncas	Incasat_Transa	Zile_Intirziere	Proc_Penaliz	Incas_Precedente	Penaliz_Transa
1111	01-08-2007	4346038	15-08-2007	53996	4	0,001	0	4346,038
1111	01-08-2007	4346038	12-03-2008	0	214	0,005	53996	21460,21
1112	01-08-2007	125516	15-08-2007	125516	4	0,001	0	125,516
1113	01-08-2007	106275	17-08-2007	106275	6	0,002	0	212,55
1114	01-08-2007	6021707	12-03-2008	0	214	0,005	0	30108,535
1115	02-08-2007	151238	12-03-2008	0	213	0,005	0	756,19
1116	02-08-2007	126713	12-03-2008	0	213	0,005	0	633,565
1117	03-08-2007	222050	16-08-2007	9754	3	0,001	0	222,05
1117	03-08-2007	222050	17-08-2007	9754	4	0,001	9754	212,296
1117	03-08-2007	222050	18-08-2007	3696	5	0,002	19508	405,084
1117	03-08-2007	222050	12-03-2008	0	212	0,005	23204	994,23
1118	04-08-2007	201975	15-08-2007	101975	1	0,0005	0	100,9875
1118	04-08-2007	201975	16-08-2007	100000	2	0,0005	101975	50

Figura 10.16. Includerea în rezultat şi a penalizărilor pentru facturile încasate parţial la data curentă

Am ajuns la capăt: includem toată interogarea precedentă într-un FROM şi aplicăm gruparea:

```
SELECT "NrFact", "DataFact", "Facturat", SUM ("Incasat_Transa") AS "Incasat",
        SUM("Penaliz_Transa") AS "Penalizari"
FROM
        (
                ... interogarea precedentă
        )
GROUP BY "NrFact", "DataFact", "Facturat"
ORDER BY 1
```

Comparând rezultatul din figura 10.17 cu cel din figura 10.16 avem motive să fim încrezători cu privire la corectitudinea rezultatului.

Este clar că situaţia clienţilor este deprimantă, însă am forţat procentele de penalizare doar pentru a vedea cum s-ar putea rezolva un asemenea gen de problemă. De-atâta oboseală, nu mai portăm interogarea în celelalte trei dialecte SQL.

NrFact	DataFact	Facturat	Incasat	Penalizari
1111	01-08-2007	4346038	53996	25806,248
1112	01-08-2007	125516	125516	125,516
1113	01-08-2007	106275	106275	212,55
1114	01-08-2007	6021707	0	30108,535
1115	02-08-2007	151238	0	756,19
1116	02-08-2007	126713	0	633,565
1117	03-08-2007	222050	23204	1833,66
1118	04-08-2007	201975	201975	150,9875
1119	07-08-2007	5774499	0	28872,495
1120	07-08-2007	97664	7315	451,745
1121	07-08-2007	4737838	0	23689,19
2111	14-08-2007	4442960	0	22214,8
2112	14-08-2007	153442	0	767,21

Figura 10.17.Centralizarea penalizărilor pe facturi

10.2. Subconsultări corelate în clauza WHERE

Corelarea subconsultărilor în clauza WHERE reprezintă una dintre cele mai provocatoare teme în SQL, fiind mai dificilă, dar şi mai generoasă, decât corelarea subconsultărilor din clauza SELECT. Este momentul intrării în scenă a unuia dintre cei mai puternici şi temuţi operatori - EXISTS.

10.2.1. Operatorul EXISTS

Ca şi IN, operatorul EXISTS permite legarea frazei SELECT principale de una sau mai multe subconsultări şi, astfel, formularea unor interogări complexe. Prin EXISTS se defineşte un predicat care are valoarea logică „adevărat" dacă subconsultarea s-a concretizat într-o tabelă ce are cel puţin o linie.

În general, EXISTS poate înlocui cu succes operatorul IN. Reciproca nefiind valabilă ([Date 1986], p. 144), unii autori, precum C.J. Date îl preferă şi recomandă formularea subconsultărilor. Pe de altă parte, însă, EXISTS este unul dintre cei mai dificili operatori. Logica sa se deosebeşte radical de cea „clasică" a lui IN.

Utilizând subconsultări necorelate, „conectate" la interogarea principală prin IN (sau =, >=, <, <=, <>), succesiunea paşilor se derulează astfel:

1. se execută subconsultarea, rezultatul fiind un set de zero, una sau mai multe linii;
2. din interogarea principală se extrag acele linii pentru valorile atributului/expresiei care sunt egale (sau diferite, mai mari ş.a.m.d. - depinde de condiţie) cu/decât cele obţinute în rezultatul subconsultării.

La utilizarea lui EXISTS, se extrag tot linii ale tabelei/tabelelor frazei SELECT principale, dar pe baza existenței a cel puțin o linie în rezultatul subconsultării; în plus, în predicatul subconsultării se face referire și la tupluri din interogarea principală (eu nu am înțeles nimic din fraza aceasta, dar dumneavoastră?).

Predicatele din clauza WHERE operează la nivel de linie, și nu la nivel de ansamblu (seturi de înregistrări), ceea ce, după Joe Celko, este o deviere de la spiritul relațional. Pe de altă parte, Sheryl Larsen ([Larsen 1996]) pledează pentru folosirea lui EXISTS în detrimentul IN-ului pe considerentul optimizării[4]. IN presupune crearea unor tabele temporare ce, uneori, reclamă timp de execuție sensibil mai mare prin comparație cu logica *tuplu-cu-tuplu* EXISTS-ențială.

Pentru comparație, începem cu exemple formulate în capitolul precedent la prezentarea operatorului IN.

Ce facturi au fost emise în aceeași zi cu factura 1120?

Varianta de interogare bazată pe operatorul EXISTS se prezintă astfel:

```
SELECT *
FROM facturi f1
WHERE EXISTS
        (SELECT *
        FROM facturi f2
        WHERE f2.NrFact = 1120 AND f1.DataFact = f2.DataFact)
```

Logica execuției se derulează după un scenariu top-bottom-top (sus-jos-sus) prezentat în figura 10.18.

Elementul de dificultate ține de modalitatea de exprimare a condiției, care este una indirectă. Se parcurge fiecare linie din SELECT-ul principal, examinându-se, pe rând, dacă există măcar o linie în subconsultare care să satisfacă predicatul subconsultării. Dacă da, atunci nu aceasta se inserează în rezultat, ci linia din SELECT-ul principal! Nu știu cum e mai nimerit să-i zicem lucru în echipă sau însușirea de către SELECT-ul principal a meritelor SELECT-ului „subaltern".

4. Cred, însă, că optimizarea e o chestiune internă a fiecărui SGBD. Deși unele variante de interogări SQL sunt, oricum, mari consumatoare de resurse, indiferent de serverul de baze de date pe care s-ar executa, există însă și diferențe semnificative de la un SGBD la altul în materie de performanțe.

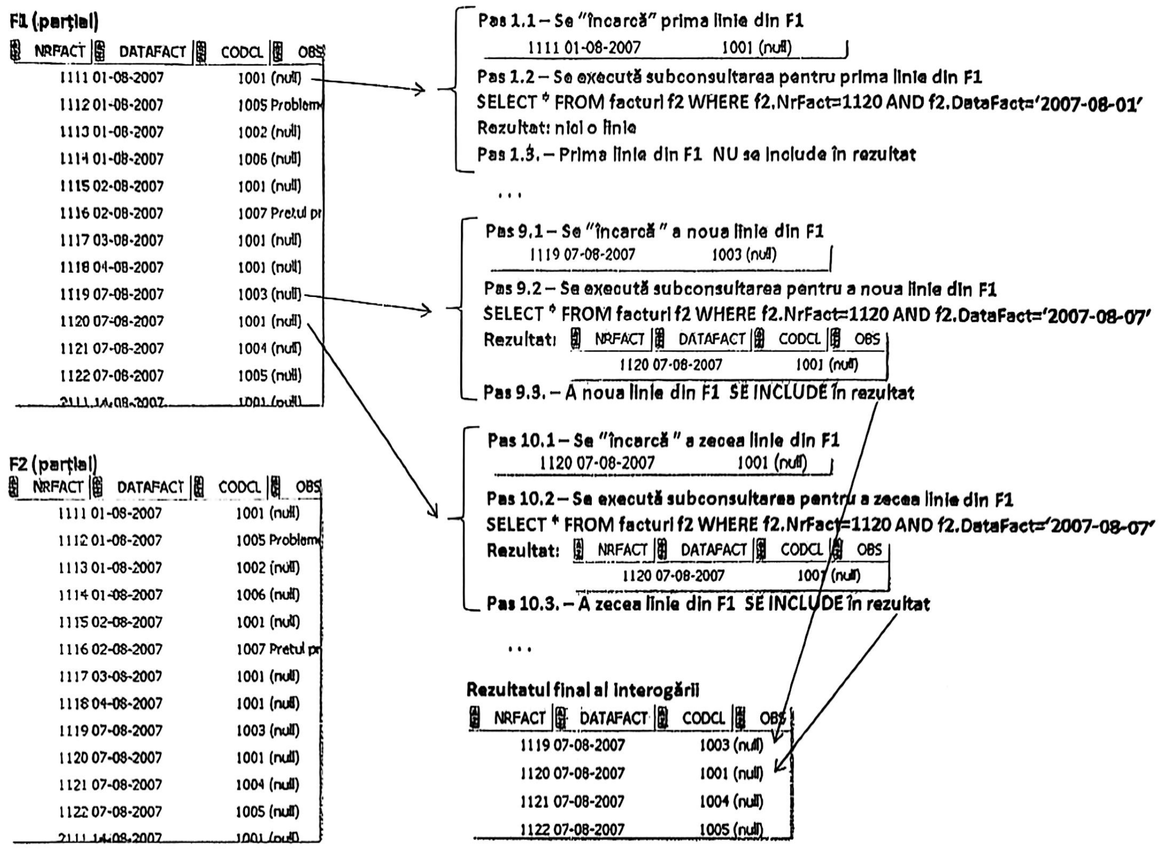


Figura 10.18. Logica operatorului EXISTS

Dacă se doreşte eliminarea din rezultat şi a facturii de referinţă, 1120, se modifică uşor interogarea:

```
SELECT *
FROM facturi f1
WHERE NrFact <> 1120 AND
      EXISTS
            (SELECT DataFact
            FROM facturi f2
            WHERE f2.NrFact = 1120 AND f1.DataFact = f2.DataFact)
```

Interesant este că interogările corelate pot fi legate nu numai prin operatorul EXISTS, ci şi de IN. Interogarea următoare este identică, în ceea ce priveşte rezultatul, cu anterioara:

```
SELECT *
FROM facturi f1
WHERE NrFact <> 1120 AND
      DataFact IN
            (SELECT DataFact
            FROM facturi f2
            WHERE NrFact = 1120 AND f1.DataFact = f2.DataFact)
```

Ce facturi au fost emise în alte zile decât factura 1120?

Următoarele două soluţii sunt echivalente:

```
SELECT *
FROM facturi f1
WHERE NOT EXISTS
        (SELECT *
        FROM facturi f2
        WHERE f2.NrFact = 1120 AND f1.DataFact = f2.DataFact)
```

şi

```
SELECT *
FROM facturi f1
WHERE EXISTS
        (SELECT *
        FROM facturi f2
        WHERE f2.NrFact = 1120 AND f1.DataFact <> f2.DataFact)
```

În subconsultarea corelată (cea care succedă operatorul EXISTS), clauza SELECT poate conţine ca argument „*" sau un atribut care, de preferinţă, nu are valori nule. Cea mai simplă, sigură şi rapidă (pentru SBGD) variantă este însă folosirea unei constante. Spre exemplu, ultima variantă se poate schimba, aproape insesizabil, în:

```
SELECT *
FROM facturi f1
WHERE EXISTS
        (SELECT 1
        FROM facturi f2
        WHERE f2.NrFact = 1120 AND f1.DataFact <> f2.DataFact)
```

Înlocuirea asteriscului sau a oricărui alt atribut cu 1 nu modifică în nici un fel corectitudinea interogării şi este de bun augur pentru simplitate.

Care sunt clienţii cărora li s-au trimis facturi în aceeaşi zi în care a fost întocmită factura 1120?

```
SELECT DenCl
FROM clienti c1
WHERE EXISTS
        (SELECT 1
        FROM facturi f1
        WHERE f1.CodCl = c1.CodCl AND EXISTS
                        (SELECT 1
                        FROM facturi f2
                        WHERE f2.NrFact = 1120 AND
                                f1.DataFact = f2.DataFact)

        )
```

Soluţia conţine două nivele de subconsultare, însă corelarea este simplă, fiecare „etaj" fiind corelat numai la nivelul imediat superior. Peste numai câteva pagini vom discuta câteva aspecte privind dubla corelare.

În ce judeţe s-a vândut produsul „Produs 2"?

```
SELECT Judet
FROM judete j
WHERE EXISTS
        (SELECT 1
        FROM coduri_postale cp
        WHERE cp.Jud = j.Jud AND EXISTS
                (SELECT 1
                FROM clienti c
                WHERE c.CodPost = cp.CodPost AND EXISTS
                        (SELECT 1
                        FROM facturi f
                        WHERE f.CodCl = c.CodCl AND EXISTS
                                (SELECT 1
                                FROM liniifact lf
                                WHERE lf.NrFact = f.NrFact AND EXISTS
                                        (SELECT 1
                                        FROM produse p
                                        WHERE p.CodPr = lf.CodPr AND
                                                DenPr = 'Produs 2'
                                        )))))
ORDER BY 1
```

Această soluţie înlocuieşte atât joncţiunea, cât şi numeroasele tabele intermediare presupuse de subconsultările necorelate.

Prin EXISTS, fireşte, se poate realiza şi intersecţia a două relaţii. Dacă luăm în discuţie relaţiile R1 şi R2 din paragraful 4.3.2 (figura 4.4), intersecţia acestora se realizează şi astfel:

```
SELECT *
FROM r1
WHERE EXISTS
        (SELECT 1
        FROM r2
        WHERE r1.A = r2.C AND r1.B = r2.D AND r1.C = r2.E)
```

În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"?

```
SELECT DISTINCT DataFact
```

```
FROM produse p1
        INNER JOIN liniifact lf1 ON p1.CodPr = lf1.CodPr
        INNER JOIN facturi f1 ON lf1.NrFact = f1.NrFact
WHERE DenPr = 'Produs 1' AND EXISTS
        (SELECT 1
        FROM produse p2
                INNER JOIN liniifact lf2 ON p2.CodPr = lf2.CodPr
                INNER JOIN facturi f2 ON lf2.NrFact = f2.NrFact
        WHERE p2.DenPr = 'Produs 2' AND f2.DataFact = f1.DataFact)
```

Sunt corelate două „instanțe" ale joncțiunii PRODUSE-LINIIFACT-FACTURI; prima conține liniile legate de Produs 1, iar a doua de Produs 2. Întrucât interesează zilele în care s-au vândut simultan cele două produse, atributul de corelare este DataFact.

Ce clienți au cumpărat și „Produs 2" și „Produs 3", dar nu au cumpărat „Produs 5"?

Acesta este exemplul în care sunt folosite, mână în mână, intersecția și diferența.

```
SELECT DISTINCT DenCl
FROM produse p1
        INNER JOIN liniifact lf1 ON p1.CodPr = lf1.CodPr
        INNER JOIN facturi f1 ON lf1.NrFact = f1.NrFact
        INNER JOIN clienti c1 ON f1.CodCl = c1.CodCl
WHERE DenPr = 'Produs 2' AND EXISTS
        (SELECT 1
        FROM produse p2
                INNER JOIN liniifact lf2 ON p2.CodPr = lf2.CodPr
                INNER JOIN facturi f2 ON lf2.NrFact = f2.NrFact
        WHERE DenPr = 'Produs 3' AND CodCl = c1.CodCl
        )
        AND NOT EXISTS
                (SELECT 1
                FROM produse p3
                        INNER JOIN liniifact lf3 ON p3.CodPr = lf3.CodPr
                        INNER JOIN facturi f3 ON lf3.NrFact = f3.NrFact
                WHERE DenPr = 'Produs 5' AND CodCl = c1.CodCl
                )
```

Fraza SELECT principală este corelată la două subconsultări dispuse „în linie", secvențial. S-au folosit și EXISTS și NOT EXISTS. Nici aceasta nu este o dublă corelare, deoarece avem de-a face cu un singur nivel de subconsultare.

Care sunt clienții cărora li s-au întocmit numai două facturi?

Este o problemă banală pentru care se pot formula soluții interesante. Firește, cea mai la îndemână variantă folosește o subconsultare necorelată:

```
SELECT DenCl
FROM clienti
WHERE CodCl IN
        (SELECT CodCl
        FROM facturi
        GROUP BY CodCl
        HAVING COUNT(NrFact) = 2)
```

Și mai simplă decât aceasta este soluția fără nicio subconsultare:

```
SELECT DenCl
FROM clienti INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
GROUP BY DenCl
HAVING COUNT(NrFact) = 2
```

Varianta pe care o consider cea mai interesantă nu folosește, pentru corelarea subconsultării cu fraza SELECT principală, nici EXISTS, nici IN, ci egalitatea:

```
SELECT DenCl
FROM clienti
WHERE 2 = ( SELECT COUNT(*)
        FROM facturi
        WHERE facturi.CodCl = clienti.CodCl)
```

Logica interogării, exprimată nu atât vizual, cât „în proză" este:

- Etapa 1:
  - se „încarcă" prima linie[5] din tabela CLIENȚI: *(1001, 'Client 1 SRL', 'R1001', 'Tranzitiei, 13 bis', '700505', NULL)* ;
  - se numără, cu funcția COUNT, câte linii din FACTURI au CodCl = 1001 (*SELECT COUNT(*) FROM facturi WHERE facturi.CodCl=1001*); rezultatul este 13;
  - se compară 2 din SELECT-ul principal cu rezultatul subconsultării corelate: $12 \neq 2$, deci „Client 1 SRL" *nu* se include în rezultat.
- Etapa 2:
  - se „încarcă" a doua linie din CLIENȚI: (*1002,'Client 2 SA', 'R1002', NULL, '700505', '0232212121'*);
  - se numără, cu funcția COUNT, câte linii din FACTURI au CodCl = 1002 (*SELECT COUNT(*) FROM facturi WHERE facturi.CodCl=1002*). Rezultatul este 3;
  - se compară 3 din SELECT-ul principal cu rezultatul subconsultării corelate: $3 \neq 2$, prin urmare, nici 'Client 2 SA' *nu* se include în rezultat.
- Etapa 3:

  ...

5. Nimic nu garantează că, în realitate, aceasta ar fi ordinea de încărcare, dar, de dragul exemplului, vom face această supoziție.

- Etapa 4:
  - se „încarcă” a patra linie din CLIENȚI: (*1004, 'Client 4', NULL, 'Sapientei, 56', '701150', NULL*);
  - se numără, cu funcția COUNT, câte linii din FACTURI au CodCl = 1004 (*SELECT COUNT(*) FROM facturi WHERE facturi.CodCl=1004*). Rezultatul este 2;
  - se compară 2 din SELECT-ul principal cu rezultatul subconsultării corelate: 2 = 2, prin urmare, „Client 4” *va fi inclus* în rezultat.

...

Analog etapele 5, 6 şi 7 (avem doar 7 clienți în tabela cu acelaşi nume).

....

Rezultatul final este cel din figura 10.19.

DENCL
Client 4

Figura 10.19. Singurul client cu două facturi

Care sunt cele mai mari cinci prețuri unitare la care s-au efectuat vânzări?

Această problemă a mai fost formulată în paragraful 9.2. Revenim cu o soluție nou-nouță, bazată pe subinterogări corelate, soluție care obține un rezultat identic celui din figura 9.10.

```
SELECT *
FROM liniifact lf1
WHERE 5 >
        (SELECT COUNT(DISTINCT lf2.PretUnit)
        FROM liniifact lf2
        WHERE lf2.PretUnit > lf1.PretUnit)
ORDER BY PretUnit DESC, NrFact
```

Ideea este grozav de ingenioasă (păcat că nu este a mea): se parcurge linie cu linie prima instanță a tabelei LINIIFACT - LF1. Pentru fiecare linie de LF1 se numără în a doua instanță a LINIIFACT - LF2 câte linii prezintă prețul unitar mai mare decât cel de pe linia curentă din LF1. Dacă numărul calculat prin *COUNT(DISTINCT lf2.PrețUnit)* este mai mic decât 5, atunci în rezultat se extrage linia curentă din LF1.

Cum nici eu n-am înțeles prea bine explicațiile de mai sus, să analizăm interogarea mai pe îndelete.

Pasul 1: se ia în discuție primul tuplu din LINIIFACT (LF1)[6]. PrețUnit are valoarea 1000. Se calculează numărul valorilor distincte ale PrețUnit din LF2

6. Ordinea în care se parcurg tuplurile din LINIIFACT (instanța LF1) nu are nici cea mai mică importanță. Pentru a nu compromite iremediabil explicațiile, vom presupune că tuplurile sunt ordonate după valorile atributelor NrFact şi Linie.

pentru care *LF2.PretUnit > LF1.PretUnit*, adică se execută interogarea: *SELECT COUNT(DISTINCT lf2.PretUnit) FROM liniifact lf2 WHERE lf2.PretUnit > 1000*. Rezultatul este 8. Prin urmare, în LINIIFACT există opt preţuri unitare peste 1000. Întrucât 8 > 5, primul tuplu din LF1 *nu* este inclus în rezultat.

Pasul 2: se „încarcă" al doilea tuplu din LINIIFACT (LF1). PreţUnit are valoarea 1050. Funcţia COUNT din subconsultare (*SELECT COUNT(DISTINCT lf2.PretUnit) FROM liniifact lf2 WHERE lf2.PretUnit > 1050*) „returnează" 9 - în LINIIFACT există nouă linii cu preţuri unitare peste 1050. Cum 9 > 5, nici al doilea tuplu din LF1 *nu este inclus* în rezultat.

Pasul 3: se „încarcă" al treilea uplu din LINIIFACT (LF1). PreţUnit este 7060. COUNT întoarce 1; 1 < 5, deci al treilea tuplu din LF1 *este inclus* în rezultatul final.

...

În total sunt 56 de paşi, corespunzători tuplurilor din LINIIFACT.

În care facturi există linii neconsecutive?

În niciuna, decocamdată. Haideţi, însă, să introducem trei facturi noi în care liniile să fie „presărate" cu destule goluri între ele. Pentru scurtarea discuţiei ne vom limita la sintaxa Oracle/PostgreSQL.

```
INSERT INTO facturi (nrfact, datafact, codcl) VALUES (4111, DATE'2007-10-01', 1001);
INSERT INTO facturi (nrfact, datafact, codcl, obs)
        VALUES (4112, DATE'2007-10-01', 1005, 'Probleme cu transportul');
INSERT INTO facturi (nrfact, datafact, codcl) VALUES (4113, DATE'2007-10-02', 1002);


INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4111, 1, 1, 57, 1000);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4111, 3, 2, 79, 1050);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4111, 5, 5, 510, 7060);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4112, 2, 2, 85, 1030);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4112, 5, 3, 65, 750);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4113, 2, 2, 120, 975);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
        VALUES (4113, 3, 1, 110, 925);
```

În factura 4111 lipsesc liniile 2 şi 4, în 4112 liniile 1, 3 şi 4, iar în 4113 linia 1. Bizuindu-ne pe interogările corelate şi operatorul EXISTS, putem formula o soluţie destul de simplă:

```
SELECT *
FROM liniifact lf1
WHERE linie > 1 AND NOT EXISTS
        (SELECT 1
        FROM liniifact lf2 WHERE NrFact = lf1.NrFact AND linie = lf1.linie-1)
ORDER BY Nrfact, Linie
```

În SELECT-ul principal (LF1) sunt luate în „discuţie" liniile mai mari ca 1, iar, dintre acestea se extrag, prin intermediul subconsultării, cele care nu au linia precedentă (adică, dacă linia curentă este 2, se verifică dacă există, în aceeaşi factură, linia 1; dacă linia curentă este 3, se verifică dacă, în aceeaşi factură, există linia 2 ş.a.m.d.). Rezultatul este corect (vezi figura 10.20).

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
4111	3	2	79	1050
4111	5	5	510	7060
4112	2	2	85	1030
4112	5	3	65	750
4113	2	2	120	975

Figura 10.20. Linii din facturi care nu au „precedent"

Dacă am fi fost mai modeşti, adică ne-ar fi interesat doar facturile în care liniile sunt „şvaiţer", puteam recurge şi la varianta:

```
SELECT NrFact
FROM liniifact
GROUP BY NrFact
HAVING COUNT(*) <> MAX(Linie)
```

Revenim la conţinutul „standard" al celor două tabele, ştergând liniile proaspăt adăugate:

```
DELETE FROM liniifact WHERE NrFact >= 4111;
DELETE FROM facturi WHERE NrFact >= 4111;
```

10.2.2. Corelări şi diviziuni

Din interogările menite să ilustreze folosirea operatorului EXISTS nu puteau lipsi cele legate de diviziunea relaţională. Începem cu un prim exemplu (teoretic) din paragraful 4.4.7. Pentru a afla valorile lui X aflate în RD1 în combinaţie cu toate valorile lui Y din RD2 se poate recurge şi la soluţia:

```
SELECT DISTINCT X
FROM rd1 T1_1
WHERE EXISTS
        (SELECT 1
        FROM rd1 T1_2
        WHERE T1_2.X = T1_1.X
```

```
        GROUP BY T1_2.X
        HAVING COUNT(DISTINCT T1_2.Y) =
                (SELECT COUNT(Y)
                FROM rd2)
)
```

Se parcurge, pe rând, fiecare linie din RD1. De fiecare dată, se verifică dacă, pentru X-ul curent, în RD1 apar toţi igrecii din RD2, verificare realizată prin GROUP BY şi HAVING.

Ce produse au fost vândute tuturor clienţilor? (exemplul 26, paragraful 4.4.7)

Pentru fiecare produs din portofoliul firmei (fiecare linie din tabela PRODUSE) se verifică dacă există linia care s-ar obţine numai dacă numărul clienţilor la care a fost vândut ar fi egal cu numărul înregistrărilor din tabela CLIENŢI:

```
SELECT DISTINCT DenPr
FROM produse p1
WHERE EXISTS (
        SELECT 1
        FROM liniifact lf
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE lf.CodPr = p1.CodPr
        GROUP BY CodPr
        HAVING COUNT(DISTINCT CodCl) =
                                (SELECT COUNT(*) FROM clienti)
                )
```

Care sunt clienţii pentru care există cel puţin câte o factură emisă în fiecare zi cu vânzări din perioada 10-30 septembrie 2007? (exemplul 23, paragraful 4.4.7)

```
SELECT DISTINCT DenCl
FROM clienti
WHERE EXISTS (
        SELECT 1
        FROM facturi
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
                AND CodCl = clienti.CodCl
        GROUP BY CodCl
        HAVING COUNT(DISTINCT DataFact) =
                (SELECT COUNT(DISTINCT DataFact)
                FROM facturi
                WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
                )
                        )
```

Ultima consultare (de jos), cea din clauza HAVING, calculează numărul total al zilelor în care sunt vânzări în intervalul 10-30 septembrie, număr care este 2. Subconsultarea „mijlocie" extrage o linie (şi o coloană) numai dacă numărul de zile de vânzări pentru clientul curent (clienti.CodCl) este egal cu 2 (rezultatul subconsultării de jos).

Care sunt facturile ce conțin măcar produsele din factura 1117? - exemplul 27 din paragraful 4.4.7.

```
SELECT DISTINCT NrFact
FROM facturi
WHERE EXISTS (
        SELECT 1
        FROM liniifact
        WHERE CodPr IN (SELECT CodPr FROM liniifact WHERE NrFact = 1117)
                AND NrFact = facturi.NrFact
        GROUP BY NrFact
        HAVING COUNT(DISTINCT CodPr) =
                (SELECT COUNT(DISTINCT CodPr)
                FROM liniifact
                WHERE NrFact = 1117)
                )
```

Atmosfera din acest paragraf devenise lâncedă, aproape plictisitoare, când am observat accidental că, în Oracle, o eroare importantă scapă neobservată (şi nepedepsită). La prima rulare, interogarea de mai sus conținea o greşeală în ultima subconsultare. În loc de

SELECT COUNT(DISTINCT CodPr) FROM *liniifact* **WHERE NrFact = 1117**

inițial era

SELECT COUNT(DISTINCT CodPr) FROM *facturi* **WHERE NrFact = 1117.**

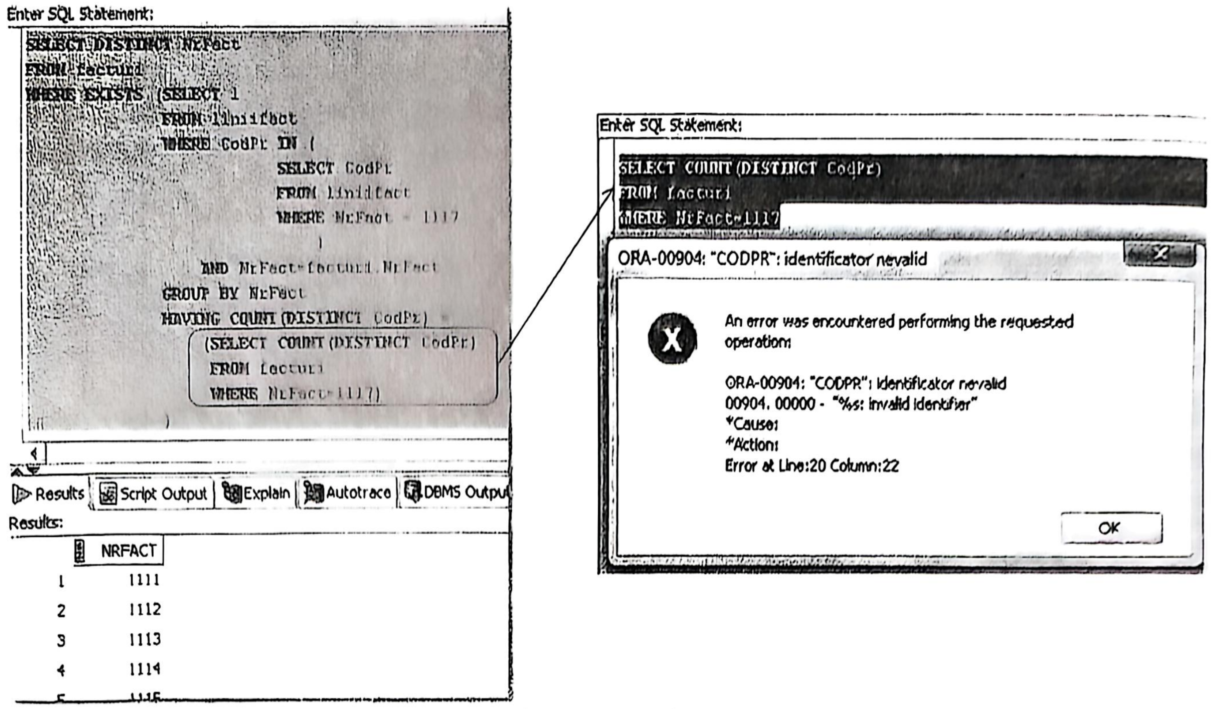


Figura 10.21. Alte surprize, suprize

Eroarea este grosolană deoarece tabela FACTURI nu are nici în clin, nici în mânecă, cu atributul CodPr, iar dacă am executa „singură" această subconsultare, am recepţiona în plină figură un meritat mesaj de mustrare (vezi partea dreaptă a figurii 10.21). Ei bine, consultarea noastră „mare" se execută fără mustrări de conştiinţă (vezi partea stângă a figurii 10.21), rezultatul fiind o brambureală.

Pentru ca lucrurile să fie şi mai interesante, aceeaşi problemă apare şi în PostgreSQL, DB2 şi în MS SQL Server. Aşa că mă întreb dacă chiar este o eroare (ştiţi gluma cu numărul oamenilor care îţi spun că eşti beat). Trecem peste nodul comun din papura celor de la IBM, Oracle, PostgreSQL şi Microsoft, şi exemplificăm corelarea fără EXISTS, luând în seamă o problemă aparent ciudată:

Care sunt produsele vândute la preţuri unitare superioare cu 14% mediei grupei din care fac parte, care sunt preţurile respective şi facturile în care apar?

Tabela PRODUSE conţine, printre altele, şi un atribut tratat cu oarecare răceală până acum - Grupa. Aflăm media preţurilor unitare dintr-o grupă şi, apropiindu-ne de enunţul problemei, media preţurilor plus 14%, prin:

```
SELECT Grupa,
        TRUNC(AVG(PretUnit),2) AS "Media",
        TRUNC(AVG(PretUnit) * 1.14,2) AS "Media + 14%"
FROM liniifact lf1 INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
GROUP BY Grupa
```

Rezultatul (vezi figura 10.22) ne interesează doar pentru a verifica dacă interogarea următoare este corectă. Oricum sintaxa era valabilă numai în Oracle şi PostgreSQL, în timp ce următoarea este valabilă în toate cele patru dialecte.

GRUPA	Media	Media + 14%
Dulciuri	1483,75	1691,47
Bere	968,82	1104,45
Tigari	3884	4427,76

Figura 10.22. Media prețurilor unitare de vânzare calculată pe grupe de produse

```
SELECT Grupa, DenPr, lf1.*
FROM liniifact lf1 INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
WHERE PretUnit >
        (SELECT AVG(PretUnit)
        FROM liniifact lf2 INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
        WHERE p2.Grupa = p1.Grupa
        ) * 1.14
ORDER BY Grupa, DenPr, PretUnit DESC
```

Ne interesează prețurile unitare ale produselor, dar şi facturile în care apar, aşa că liniile după care se face corelarea sunt cele ale joncțiunii LINIIFACT-PRODUSE. Noutatea, în economia acestui paragraf, o aduce operatorul de corelare dintre consultarea principală şi subconsultarea ei, operator care este >. Cum clauza SELECT a subconsultării conține funcția-agregat AVG (subconsultarea este scalară), interogarea funcționează - vezi figura 10.23.

GRUPA	DENPR	NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT
Bere	Produs 2	1120	1	2	80	1120
Dulciuri	Produs 4	1114	2	4	30	1705
Tigari	Produs 5	2121	1	5	550	7064
Tigari	Produs 5	1114	3	5	700	7064
Tigari	Produs 5	1121	1	5	550	7064
Tigari	Produs 5	1111	3	5	500	7060
Tigari	Produs 5	2111	3	5	510	7060
Tigari	Produs 5	3111	3	5	510	7060
Tigari	Produs 5	3119	4	5	755	6300
Tigari	Produs 5	2119	4	5	755	6300
Tigari	Produs 5	1119	4	5	750	6300

Figura 10.23. Produse vândute la prețuri superioare cu mai mult de 14% decât media grupei din care fac parte

10.3. Alte tipuri de corelări simple

Care este evoluția zilnică a vânzărilor, raportat la ziua de vânzări anterioară?

În problema din paragraful 10.1 (vezi şi figura 10.6), diferența era calculată între ziua curentă şi ziua calendaristică precedentă. Astfel încât, toate zilele de luni erau raportate la zero, deoarece, pentru cea mai mare parte a firmelor, duminica nu se

lucrează. Acum dorim ca diferenţa să fie calculată între vânzările din ziua curentă şi cele din precedenta zi de vânzări, adică între luni şi vineri ş.a.m.d., după cum se observă în figura 10.24.

ZI	Vinzari_Zi_Curenta	Vinzari_Zi_Precedenta	Diferenta
01-08-2007	10599535	0	10599535
02-08-2007	277950	10599535	-10321585
03-08-2007	222050	277950	-55900
04-08-2007	201975	222050	-20075
07-08-2007	10610000,5	201975	10408025,5
14-08-2007	4723931,5	10610000,5	-5886069
15-08-2007	247757	4723931,5	-4476174,5
16-08-2007	467420	247757	219663
21-08-2007	10560939,5	467420	10093519,5
01-09-2007	4596401,5	10560939,5	-5964538
02-09-2007	238437,5	4596401,5	-4357964
07-09-2007	5819668	238437,5	5581230,5
10-09-2007	424704,5	5819668	-5394963,5
17-09-2007	179565	424704,5	-245139,5

Figura 10.24. Diferenţele dintre două zile consecutive de vânzări

```
SELECT DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS "Vanzari_Zi_Curenta ",
        (SELECT COALESCE(SUM(Cantitate * PretUnit * (1 + ProcTVA)) ,0)
        FROM facturi f2
                INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
        WHERE DataFact IN (
                SELECT MAX(DataFact)
                FROM facturi f3
                        INNER JOIN liniifact lf3 ON f3.NrFact = lf3.NrFact
                        INNER JOIN produse p3 ON lf3.CodPr = p3.CodPr
                WHERE DataFact < f.DataFact AND
                                (Cantitate * PretUnit * (1 + ProcTVA)) > 0
                                )
        ) AS "Vanzari_Zi_Precedenta ",
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) - COALESCE(
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                FROM facturi f2
                        INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
                        INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
```

```
            WHERE DataFact IN (
                    SELECT MAX(DataFact)
                    FROM facturi f3
                            INNER JOIN liniifact lf3 ON f3.NrFact = lf3.NrFact
                            INNER JOIN produse p3 ON lf3.CodPr = p3.CodPr
                    WHERE DataFact < f.DataFact AND
                                    (Cantitate * PretUnit * (1 + ProcTVA)) > 0
                            )
                                            ),0) AS "Diferenta "
FROM facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY DataFact
ORDER BY DataFact
```

Artificiul necesar extragerii zilei precendente de vânzări se găseşte în subconsultările din cele două interogări scalare. Prin joncţiunea instanţelor F3 şi LF3 ale tabelelor FACTURI şi LINIIFACT şi condiţia *DataFact < F.DataFact,* vor fi extrase toate liniile din facturi întocmite înaintea datei curente (F este instanţa principală a tabelei FACTURI - cea care indică ziua curentă). Ca măsură suplimentară de precauţie se verifică dacă *Cantitate * PreţUnit * (1 + ProcTVA)) > 0* (nu cumva ca să fie vreo zi în care apare o factură în care liniile să prezinte Cantitate = 0, deşi situaţia aceasta este greu de imaginat). Dintre zilele de vânzare ce precedă ziua curentă, cea mai apropiată (calendaristic) se extrage prin funcţia MAX.

Noutatea acestei soluţii ţine de corelarea, de data aceasta, a frazei SELECT principale cu subconsultarea subconsultării scalare (cea din clauza SELECT) (vezi figura 10.25). Nu ştiu ce ziceţi dumneavoastră, dar mie-mi place...

```
SELECT DataFact AS Zi,
        SUM(Cantitate * PretUnit * (1+ProcTVA))  AS "Vinzari_Zi_Curenta",
        (SELECT COALESCE(SUM(Cantitate * PretUnit * (1+ProcTVA)) ,0)
         FROM facturi f2
                INNER JOIN liniifact lf2 ON f2.NrFact=lf2.NrFact
                INNER JOIN produse p2 ON lf2.CodPr=p2.CodPr
         WHERE DataFact  IN (
                SELECT MAX(DataFact)
                FROM  facturi f3
                        INNER JOIN liniifact lf3 ON f3.NrFact=lf3.NrFact
                        INNER JOIN produse p3 ON lf3.CodPr=p3.CodPr
                WHERE DataFact < f.DataFact AND
                              (Cantitate * PretUnit * (1+ProcTVA)) > 0
                              )
        ) AS "Vinzari_Zi_Precedenta",
        SUM(Cantitate * PretUnit * (1+ProcTVA)) - COALESCE(
                (SELECT SUM(Cantitate * PretUnit * (1+ProcTVA))
                 FROM facturi f2
                        INNER JOIN liniifact lf2 ON f2.NrFact=lf2.NrFact
                        INNER JOIN produse p2 ON lf2.CodPr=p2.CodPr
                WHERE DataFact IN (
                        SELECT MAX(DataFact)
                        FROM  facturi f3
                                INNER JOIN liniifact lf3 ON f3.NrFact=lf3.NrFact
                                INNER JOIN produse p3 ON lf3.CodPr=p3.CodPr
                        WHERE DataFact < f.DataFact AND
                                (Cantitate * PretUnit * (1+ProcTVA)) > 0
                                )
                                                                ),0)  AS "Diferenta"
FROM facturi f
        INNER JOIN liniifact lf ON f.Nrfact=lf.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
GROUP BY DataFact
ORDER BY DataFact
```

Figura 10.25. Corelarea liniilor tabelei FACTURI din SELECT-ul principal cu o subconsultare a subconsultării scalare

Care sunt facturile cu valori mai mari decât media zilei în care au fost întocmite?

E timpul unui nou experiment. Încercăm să corelăm o subconsultare din clauza HAVING cu grupurile interogării principale:

```
SELECT f1.NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Valoare
FROM facturi f1
        INNER JOIN liniifact lf1 ON f1.NrFact = lf1.NrFact
        INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
GROUP BY f1.NrFact
HAVING SUM(Cantitate * PretUnit * (1 + ProcTVA)) >=
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))/
                        COUNT(DISTINCT f2.NrFact)
```

FROM facturi f2

INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact

INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr

WHERE f2.DataFact = f1.DataFact

)

Din păcate trucul nu funcţionează în niciunul dintre SGBD-urile testate - vezi cazul PostgreSQL în figura 10.26. Noroc că mesajul de eroare este unul civilizat.

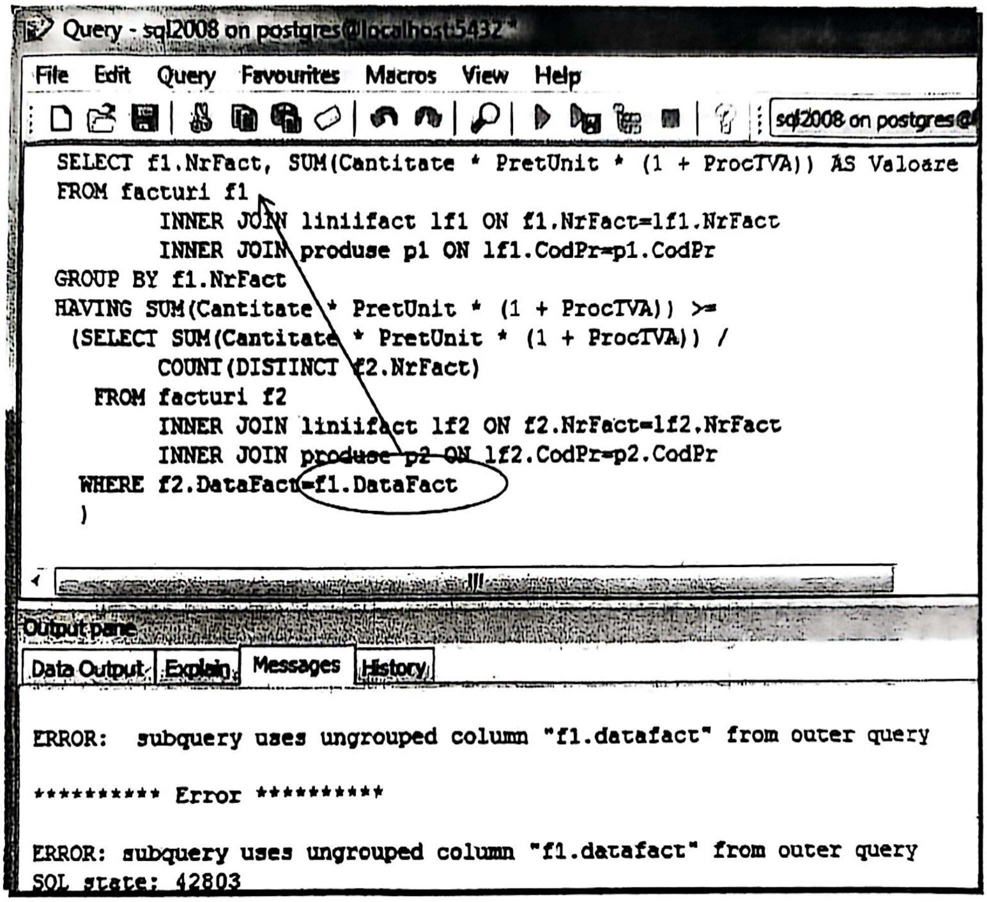


Figura 10.26. O corelare eşuată

10.4. Corelarea dublă

Dubla corelare constituie o facilitate bine ocolită de către SQL-işti (eu, unul, am reuşit s-o evit cu brio până în acest paragraf şi aşa voi face şi după). Sheryl Larsen estima în 1998 că mai puţin de 10% dintre dezvoltatorii de aplicaţii în DB2 aveau cunoştinţă de interogările dublu-corelate. Mărturisesc că în articolul lui Sheryl ([Larsen 1998]) am găsit cea mai bună explicaţie a acestui mecanism.

În cazul corelării simple, scenariul de execuţie este unul de tip top-bottom-top; la corelarea dublă lucrurile stau cu mult mai interesant: top-bottom-middle--bottom-middle-top. Mai simplu nici că se poate!

Dacă n-ar fi fost în joc diviziunea relaţională, n-am fi ajuns aici cu „hermeneutica" SQL. Revenim la cele două tabele, RD1 şi RD2, pe care le-am folosit în

capitolul 4 la prezentarea diviziunii relaţionale. Valorile lui X care se găsesc în RD1 în combinaţie cu toate valorile lui Y din RD2 pot fi aflate şi prin interogarea următoare:

```
SELECT DISTINCT X
FROM rd1 T1_1
WHERE NOT EXISTS
        (SELECT 1
        FROM rd2
        WHERE NOT EXISTS
                (SELECT 1
                FROM rd1 T1_2
                WHERE T1_2.X = T1_1.X AND T1_2.Y = rd2.Y
                )
        )
```

Execuţia frazei SQL se derulează astfel:

1. „Faza" 1:
   - 1.1. Se „încarcă" prima linie din T1_1: *(x1, y1)* şi prima linie din RD2: *y1*.
     - 1.1.1. se testează dacă există în T1_2 o linie în care *X = x1 (T1_1.X) şi Y = y1 (RD2.Y)*. Există!, astfel încât se trece la următoarea linie din RD2,
     - 1.1.2. se testează dacă există în T1_2 o linie în care *X = x1 (T1_1.X) şi Y = y2 (R2.Y)*. Există!, deci se „avansează" încă o linie în RD2,
     - 1.1.3. se testează dacă există în T1_2 o linie în care *X = x1 (T1_1.X) şi Y = y3 (R2.Y)*. Există!, deci se „avansează" încă o linie în RD2,
     - 1.1.4. se testează dacă există în T1_2 o linie în care *X = x1 (T1_1.X) şi Y = y5 (R2.Y)*. Există!, deci se „avansează" încă o linie în RD2,
     - 1.1.5. se testează dacă există în T1_2 o linie în care *X = x1 (T1_1.X) şi Y = y5 (R2.Y)*. Există!, deci se încearcă încărcarea următoarei linii în RD2,
     - 1.1.6. întrucât am ajuns la sfârşitul tabelei RD2 şi toate valorile din RD2 s-au regăsit în T1_2 în combinaţie cu x1.
   - 1.2. Subconsultarea de jos (*bottom*) întoarce TRUE către subconsultarea din mijloc (*middle*). Subconsultarea din mijloc recepţionează rezultatul SELECT-ului de jos şi îl pasează cu aceeaşi valoare logică sau cu valoare schimbată (dacă apare NOT) interogării principale. În această situaţie, SELECT-ul mijlociu primeşte de la cel de jos valoarea logică TRUE, însă, deoarece operatorul de conexiune mijloc-jos este NOT EXISTS, va întoarce SELECT-ului principal FALSE.
   - 1.3. Fraza SELECT principală recepţionează valoarea logică FALSE de la subconsultarea mijlocie. Dar operatorul de conexiune sus-mijloc este NOT EXISTS, iar FALSE-ul este transformat în TRUE, *iar valoarea x1 va fi inclusă în rezultat!*
2. „Faza" 2:
   - 2.1. se „încarcă" al doilea tuplu din T1_1: *(x2, y1)* şi primul tuplu din RD2 : *y1*

2.1.1. se testează dacă există în T1_2 o linie în care *X = x2 (T1_1.X) şi Y = y1 (RD2.Y)*. Există!, astfel încât se trece la următoarea linie din RD2,

2.1.2. se testează dacă există în T1_2 o linie în care *X = x2 (T1_1.X) şi Y = y2 (RD2.Y)*. Nu există!, iar încărcarea celorlalte linii din RD2 este abandonată şi

2.2. subconsultarea de jos (*bottom*) întoarce FALSE către subconsultarea din mijloc (*middle*). SELECT-ul mijlociu recepţionează valoarea logică FALSE, şi, datorită operatorului NOT EXISTS, „pasează" SELECT-ului principal TRUE.

2.3. Fraza SELECT principală recepţionează valoarea logică TRUE de la subconsultarea mijlocie, pe care, la rândul său, o transformă în FALSE, iar *valoarea x2 nu va fi inclusă în rezultat*!

...

Lucrurile se continuă cu încă 14 faze, una pentru fiecare linie din RD1. Clauza DISTINCT asigură preluarea în rezultat a fiecărei valori o singură dată.

Dacă privim mai îndeaproape fraza SELECT, observăm că, de fapt, aceasta răspunde la întrebarea: *pentru care dintre valorile lui X nu există nici un Y care să nu apară cu X-ul respectiv în combinaţie pe (măcar) o linie în RD1.*

În ce zile s-au vândut şi produsul cu denumirea „Produs 1" şi cel cu denumirea „Produs 2"?

Soluţia bazată pe corelare dublă este probabil cea mai complicată, mult prea complicată pentru aşa o bagatelă de problemă:

```
SELECT DISTINCT DataFact
FROM facturi f1
        INNER JOIN liniifact lf1 ON f1.NrFact = lf1.NrFact
        INNER JOIN produse p1 ON lf1.CodPr = p1.CodPr
WHERE NOT EXISTS
        (SELECT 1
        FROM produse p1
        WHERE DenPr IN ('Produs 1', 'Produs 2') AND NOT EXISTS
                (SELECT 1
                FROM facturi f2
        INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
        INNER JOIN produse p2 ON lf2.CodPr = p2.CodPr
                WHERE f2.DataFact = f1.DataFact AND p2.CodPr = p1.CodPr
                )
        )
```

Extrageţi clienţii pentru care există cel puţin câte o factură emisă în fiecare zi.

Reluăm această problemă de la diviziunea relaţională ce poate fi rezolvată şi prin SELECT-ul următor:

```
SELECT DISTINCT DenCl
FROM clienti c1 INNER JOIN facturi f1 ON c1.CodCl = f1.CodCl
```

```
WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        AND NOT EXISTS
        (SELECT 1
        FROM facturi f2
        WHERE DataFact BETWEEN DATE'2007-09-10' AND DATE'2007-09-30'
        AND NOT EXISTS
                (SELECT 1
                 FROM clienti c3 INNER JOIN facturi f3
                        ON c3.CodCl = f3.CodCl
        WHERE DataFact BETWEEN DATE'2007-09-10'
        AND DATE'2007-09-30' AND
        c3.CodCl = c1.CodCl AND f3.DataFact = f2.DataFact
        )
)
```

Toate soluţiile ce întrebuinţează dubla corelare de până acum au o concurenţă care le pune în umbră. Variantele nebazate pe dubla corelare sunt mai simple, atât ca dimensiune, cât şi ca mod de înţelegere. Este timpul să scoatem din mânecă o problemă de diviziune relaţională care să demonstreze adevărata forţă a dublei corelări şi să justifice generosul efort intelectual pentru a o înţelege:

Ce facturi conţin măcar produsele din factura 1117?

```
SELECT DISTINCT NrFact
FROM liniifact lf1
WHERE NOT EXISTS
        (SELECT 1
        FROM liniifact lf2
        WHERE NrFact = 1117
        AND NOT EXISTS
                (SELECT 1
                FROM liniifact lf3
                WHERE lf3.CodPr = lf2.CodPr AND lf3.NrFact = lf1.NrFact ) )
```

Fraza SELECT analizată răspunde, de fapt, la întrebarea: *Pentru care facturi nu există nici un produs ce apare în factura 1117, dar nu este prezent în factura respectivă?*

Fără dubla corelare, pot fi formulate şi soluţii bazate pe EXCEPT (sau minus în Oracle) şi NOT IN precum:

```
SELECT NrFact FROM liniifact
        EXCEPT
        SELECT DISTINCT lf1.NrFact FROM liniifact lf1, liniifact lf2
WHERE lf2.NrFact = 1117 AND (lf1.Nrfact, lf2.CodPr) NOT IN
        (SELECT NrFact, CodPr FROM liniifact)
```

şi (SQL Server):

```
SELECT NrFact FROM liniifact
        EXCEPT
SELECT DISTINCT lf1.NrFact FROM liniifact lf1, liniifact lf2
WHERE lf2.NrFact = 1117 AND
        CAST (lf1.Nrfact AS CHAR(8)) + CAST (lf2.CodPr AS CHAR(6)) NOT IN
                (SELECT CAST (NrFact AS CHAR(8)) + CAST (CodPr AS CHAR(6))
                FROM liniifact)
```

Care sunt clienţii cărora li s-au vândut cel puţin produsele vândute clientului CLIENT 4?

Analizând enunţul problemei, *„Care sunt clienţii?..."*, rezultă că atributul pentru corelarea consultării principale (*top*) şi celei de jos (*bottom*) este CodCl, iar din partea condiţională a enunţului *„... cel puţin produsele..."* pentru corelarea mijloc (*middle*) - jos (*bottom*) se utilizează atributul CodPr.

```
SELECT DenCl
FROM clienti c1
WHERE NOT EXISTS
        (SELECT 1
        FROM clienti c2
                INNER JOIN facturi f2 ON c2.CodCl = f2.CodCl
                INNER JOIN liniifact lf2 ON f2.NrFact = lf2.NrFact
        WHERE DenCl = 'Client 4' AND NOT EXISTS
                (SELECT 1
        FROM facturi f3
        INNER JOIN liniifact lf3 ON f3.NrFact = lf3.NrFact
                WHERE c1.CodCl = f3.CodCl AND lf3.CodPr = lf2.CodPr)
        )
```

Capitolul 11
OLAP în SQL

Cunoaştem cu toţii zicala cu fratele, necuratul şi puntea[1]. Mulţi dintre noi o aplicăm, mai ales în momentele de compromis, fireşte, fiecare imaginându-şi că celălalt e dracul. În materie de funcţii analitice ce au fost introduse ca amendament la standardul SQL:1999 puntea este ANSI (American National Standard Institute), iar cei doi parteneri principali (care o fi necuratul?) sunt IBM şi Oracle.

Derulată de pe la începutul anului 1999, colaborarea dintre echipele conduse de Hamid Paresh la IBM (Almaden Research Laboratory) şi Andy Witkowski la Oracle s-a concretizat rapid într-un set comun de specificaţii pe care l-au înaintat ANSI (de la ANSI, Jim Melton a jucat un rol important în acest proiect). În scurt timp, pe baza specificaţiilor, ANSI a publicat Amendamentul 1 la standardul SQL:1999.

IBM a implementat o parte din opţiunile OLAP ale SQL încă din versiunea 6.1 a DB2-ului, în timp ce Oracle începând cu versiunea 8i2, ambele lansate pe piaţă în anul 1999. La scurt timp, şi MS SQL Server a implementat câteva funcţii analitice. De atunci, şi versiunile standardului, şi cele ale serverelor de baze de date au evoluat continuu.

Definite uneori şi funcţii analitice, funcţiile/opţiunile OLAP (de la *On Line Analytical Processing*) extind nucleul SQL către probleme legate destul de strâns de ceea ce îndeobşte este cunoscut sub titulaturile depozite de date (Data Warehouses), Data Mining etc. Din păcate, în acest capitol, PostgreSQL-ul stă pe tuşă, neavând implementată, până la versiunea curentă (8.3), nicio funcţie OLAP.

11.1. Subtotaluri

Problema subtotalurilor în SQL a fost abordată cu mari sforţări încă din paragraful 7.2. Am convenit că, adunând mai multe fraze SELECT conectate prin UNION (plus grupările de rigoare), se poate obţine o formă rezonabilă de raport. Peste câteva rânduri o să vedem cât de mult se simplifică lucrurile folosind operatorii ROLLUP, CUBE etc.

---

1. Nu e prea elegant să te autocitezi sau să repeţi glumele, mai ales atunci când sunt slabe. Însă, pur şi simplu, nu m-am putut abţine să nu inserez introducerea din materialul publicat în PC Report-ul nr. 98 (noiembrie), 2000.

11.1.1. Operatorul ROLLUP

Să se obţină situaţia vânzărilor pe clienţi şi zile în luna septembrie 2007, afişându-se câte un subtotal la nivel de client şi un total general.

Dorim să obţinem, mai întâi, raportul din figura 7.15 pentru care în paragraful 7.2 reuneam trei SELECT-uri, unul pentru rândul „curent", altul pentru subtotalul la nivel de clienţi şi un altul pentru totalul general. Pentru asemenea gen de probleme avem operatorul ROLLUP aplicat după cum urmează:

```
SELECT DenCl AS DenumireClient, DataFact AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
```

Ceea ce se obţine în Oracle cu această interogare nu seamănă leit cu figura 7.2, însă e un bun punct de plecare (vezi figura 11.1). Clienţii nu sunt dispuşi în ordinea alfabetică a denumirii lor, însă liniile unui client sunt consecutive, iar rândul de total apare la sfârşit.

DENUMIRECLIENT	DATA	VINZARI
Client 2 SA	02-09-2007	127530
Client 2 SA	(null)	127530
Client 1 SRL	01-09-2007	4442960
Client 1 SRL	02-09-2007	110908
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 1 SRL	(null)	5021287
Client 3 SRL	07-09-2007	5819668
Client 3 SRL	(null)	5819668
Client 5 SRL	01-09-2007	153442
Client 5 SRL	(null)	153442
Client 7 SRL	10-09-2007	136850
Client 7 SRL	(null)	136850
(null)	(null)	11258777

Figura 11.1. Operatorul ROLLUP (în Oracle) fără finisaje

MS SQL Server are o sintaxă uşor diferită faţă de standardul SQL:

```
SELECT DenCl AS DenumireClient, DataFact AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
```

```
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DATEPART (YEAR,DataFact) = 2007
            AND DATEPART(MONTH,DataFact) = 9
GROUP BY DenCl, DataFact WITH ROLLUP
```

În materie de formă a rezultatului, SQL Server-ul prezintă avantajul ordonării implicite a liniilor după denumirea clientului şi data facturării (vezi figura 11.2).

DenumireClient	Data	Vinzari
Client 1 SRL	2007-09-01 00:00:00	4442960.0000
Client 1 SRL	2007-09-02 00:00:00	110908.0000
Client 1 SRL	2007-09-10 00:00:00	287855.0000
Client 1 SRL	2007-09-17 00:00:00	179565.0000
Client 1 SRL	NULL	5021287.0000
Client 2 SA	2007-09-02 00:00:00	127530.0000
Client 2 SA	NULL	127530.0000
Client 3 SRL	2007-09-07 00:00:00	5819668.0000
Client 3 SRL	NULL	5819668.0000
Client 5 SRL	2007-09-01 00:00:00	153442.0000
Client 5 SRL	NULL	153442.0000
Client 7 SRL	2007-09-10 00:00:00	136850.0000
Client 7 SRL	NULL	136850.0000
NULL	NULL	11258777.0000

Figura 11.2. ROLLUP în MS SQL Server

DB2-ul are o sintaxă similară Oracle, cu excepţia funcţiei EXTRACT:

```
SELECT ... FROM ...
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
```

Ordonarea liniilor este una specială (vezi figura 11.3), valorile NULL fiind plasate înaintea celor nenule.

DENUMIRECLIENT	DATA	VINZARI
		11.258.777,0000
Client 1 SRL		5.021.287,0000
Client 2 SA		127.530,0000
Client 3 SRL		5.819.668,0000
Client 5 SRL		153.442,0000
Client 7 SRL		136.850,0000
Client 1 SRL	01.09.2007	4.442.960,0000
Client 1 SRL	02.09.2007	110.908,0000
Client 1 SRL	10.09.2007	287.855,0000
Client 1 SRL	17.09.2007	179.565,0000
Client 2 SA	02.09.2007	127.530,0000
Client 3 SRL	07.09.2007	5.819.668,0000
Client 5 SRL	01.09.2007	153.442,0000
Client 7 SRL	10.09.2007	136.850,0000

Figura 11.3. ROLLUP în DB2

În continuare, încercăm să scăpăm de NULL-uri şi să inserăm „mesaje" lămuritoare. Cum textele de pe coloana dedicată datei facturilor ar genera o eroare de incompatilitate a tipurilor de date, concatenăm (implicit în Oracle şi explicit în SQL Server şi DB2) DataFact cu un spaţiu:

```
SELECT DenCl AS DenumireClient,
        COALESCE (CAST (DataFact AS CHAR(10)) ||
        ' ','subtotal (toate zilele)') AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
```

DENUMIRECLIENT	DATA	VINZARI
Client 2 SA	02-09-2007	127530
Client 2 SA		127530
Client 1 SRL	01-09-2007	4442960
Client 1 SRL	02-09-2007	110908
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 1 SRL		5021287
Client 3 SRL	07-09-2007	5819668
Client 3 SRL		5819668
Client 5 SRL	01-09-2007	153442
Client 5 SRL		153442
Client 7 SRL	10-09-2007	136850
Client 7 SRL		136850
(null)		11258777

Figura 11.4. Operatorul ROLLUP (în Oracle) cu finisaje defecte

Ceea ce obținem în Oracle ne derutează (un pic) (vezi figura 11.4). Descoperim, astfel, că, în Oracle, NULL concatenat cu spațiu nu e chiar NULL, ci spațiu (ne reamintim că prin adunarea unui număr cu un NULL obțineam NULL). Ciudat este că dacă portăm soluția în SQL Server:

```
SELECT DenCl AS DenumireClient,
        COALESCE (CAST (DataFact AS CHAR) + ' ','subtotal (toate zilele)') AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0,0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DATEPART (YEAR,DataFact) = 2007
                AND DATEPART(MONTH,DataFact) = 9
GROUP BY DenCl, DataFact WITH ROLLUP
```

rezultatul este altul (vezi figura 11.5).

DenumireClient	Data	Vinzari
Client 1 SRL	Sep 1 2007 12:00AM	4442960.0000
Client 1 SRL	Sep 2 2007 12:00AM	110908.0000
Client 1 SRL	Sep 10 2007 12:00AM	287855.0000
Client 1 SRL	Sep 17 2007 12:00AM	179565.0000
Client 1 SRL	subtotal (toate zilele)	5021287.0000
Client 2 SA	Sep 2 2007 12:00AM	127530.0000
Client 2 SA	subtotal (toate zilele)	127530.0000
Client 3 SRL	Sep 7 2007 12:00AM	5819668.0000
Client 3 SRL	subtotal (toate zilele)	5819668.0000
Client 5 SRL	Sep 1 2007 12:00AM	153442.0000
Client 5 SRL	subtotal (toate zilele)	153442.0000
Client 7 SRL	Sep 10 2007 12:00AM	136850.0000
Client 7 SRL	subtotal (toate zilele)	136850.0000
NULL	subtotal (toate zilele)	11258777.0...

Figura 11.5. Un alt tratament al concatenării spaţiilor cu NULL-i în MS SQL Server

DB2 se comportă similar SQL Server-ului, cu diferenţă că, în rezultat, linia de total general apare nu la sfârşit, ci la începutul listei. După aceasta urmează liniile de subtotal şi apoi cele „curente". În consecinţă, Oracle ne dă ghes să folosim o structură CASE:

```
SELECT COALESCE(DenCl, '{Total clienti}') AS DenumireClient,
        CASE WHEN DataFact IS NULL THEN 'subtotal (toate zilele)'
        ELSE CAST (DataFact AS CHAR(10)) || ' ' END AS Data,
    ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
    INNER JOIN facturi f ON c.CodCl = f.CodCl
    INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
    INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
    AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
```

Figura 11.6 ne arată că facem progrese remarcabile, însă la rândul de total general, pe care apare acum *{Total clienţi}*, pe coloana Data apare şirul *subtotal (toate zilele)*, ceea ce, iarăşi, ne irită un pic (nu ştiu de ce suntem aşa de iritaţi astăzi!).

DENUMIRECLIENT	DATA	VINZARI
Client 2 SA	02-09-2007	127530
Client 2 SA	subtotal (toate zilele)	127530
Client 1 SRL	01-09-2007	4442960
Client 1 SRL	02-09-2007	110908
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 1 SRL	subtotal (toate zilele)	5021287
Client 3 SRL	07-09-2007	5819668
Client 3 SRL	subtotal (toate zilele)	5819668
Client 5 SRL	01-09-2007	153442
Client 5 SRL	subtotal (toate zilele)	153442
Client 7 SRL	10-09-2007	136850
Client 7 SRL	subtotal (toate zilele)	136850
{Total clienti}	subtotal (toate zilele)	11258777

Figura 11.6. Un ghimpe Oracle la rândul de total general

În plus, nici acum clienții nu sunt ordonați alfabetic, așa că, imaginăm o nouă versiune în care să includem un CASE în CASE, plus o clauză de ordonare:

```
SELECT COALESCE(DenCl, '{TOTAL GENERAL}') AS DenumireClient,
        CASE
                WHEN DataFact IS NULL THEN
                        CASE WHEN DenCl IS NULL THEN ' '
                        ELSE 'subtotal (toate zilele)'
                        END
                ELSE CAST (DataFact AS CHAR(10)) || ' '
        END AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
ORDER BY 1,2
```

După „scrutarea" figurii 11.7 este evident că în Oracle avem o zi proastă, întrucât chiar dacă rândul total este în regulă, clienții sunt ordonați după denumire, liniile fiecărui client sunt ordonate aiurea, rândul de subtotal fiind plasat la începutul clientului, nu la final.

DENUMIRECLIENT	DATA	VINZARI
Client 1 SRL	subtotal (toate zilele)	5021287
Client 1 SRL	01-09-2007	4442960
Client 1 SRL	02-09-2007	110908
Client 1 SRL	10-09-2007	287855
Client 1 SRL	17-09-2007	179565
Client 2 SA	subtotal (toate zilele)	127530
Client 2 SA	02-09-2007	127530
Client 3 SRL	subtotal (toate zilele)	5819668
Client 3 SRL	07-09-2007	5819668
Client 5 SRL	subtotal (toate zilele)	153442
Client 5 SRL	01-09-2007	153442
Client 7 SRL	subtotal (toate zilele)	136850
Client 7 SRL	10-09-2007	136850
{TOTAL GENERAL}		11258777

Figura 11.7. Alt ghimpe Oracle (poziţia liniei de subtotal la nivel de client)

Necazul este cu atât mai mare, cu cât MS SQL Server ordonează liniile în rezultat cum trebuie, chiar şi fără clauza ORDER BY:

```
SELECT COALESCE(DenCl, '{TOTAL GENERAL}') AS DenumireClient,
        CASE
                WHEN DataFact IS NULL THEN
                        CASE
                                WHEN DenCl IS NULL THEN ' '
                                ELSE 'subtotal (toate zilele)'
                        END
                ELSE CAST (DataFact AS CHAR) + ' '
        END AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0,0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DATEPART (YEAR,DataFact) = 2007
                AND DATEPART(MONTH,DataFact) = 9
GROUP BY DenCl, DataFact WITH ROLLUP
```

Obţinerea ordinii dorite în Oracle nu necesită un efort intelectual deosebit. Este suficient că în CASE-ul „principal", pe ramura ELSE, să adăugăm un spaţiu şi înaintea DataFact (vezi textul italic din interogarea de mai jos:

```
SELECT COALESCE(DenCl, '{TOTAL GENERAL}') AS DenumireClient,
        CASE
                WHEN DataFact IS NULL THEN
```

```
                    CASE WHEN DenCl IS NULL THEN ' '
                    ELSE 'subtotal (toate zilele)'
                    END
              ELSE ' ' || DataFact || ' '
        END AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA))) AS Vanzari
FROM
  -- ... continuare identică interogării Oracle precedente
```

Nu mai includem figura cu rezultatul, dar credeți-mă pe cuvânt că ordinea este așa cum ne-am propus-o. DB2 face nazuri similare, chiar și cu rândul de total general, așa că avem toate motivele să folosim varianta:

```
SELECT COALESCE(' '|| DenCl, '{TOTAL GENERAL}') AS DenumireClient,
        CASE
              WHEN DataFact IS NULL THEN
                    CASE WHEN DenCl IS NULL THEN ' '
                    ELSE 'subtotal (toate zilele)'
                    END
        ELSE ' ' || CAST(DataFact AS CHAR(10)) || ' '
        END AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
GROUP BY ROLLUP (DenCl, DataFact)
ORDER BY 1,2
```

Care este totalul vânzărilor, pe clienți, localități, județe și totalul general?

Pe baza soluției de la problema precedentă, bănuiam că nu mai avem ce comenta:

```
SELECT COALESCE(Judet, '{TOTAL GENERAL}') AS Judet,
        CASE  WHEN Judet IS NULL THEN ' '
              ELSE COALESCE(Loc, '{Subtotal-JUDET}' || Judet)
        END AS Loc,
        CASE  WHEN Loc IS NULL THEN ' '
              ELSE COALESCE(DenCl, '{Subtotal-LOCALITATE}' || Loc)
        END AS DenCl,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
```

```
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY ROLLUP (Judet, Loc, DenCl)
ORDER BY 1,2,3
```

Şi în acest caz (Oracle) ordinea dispunerii liniilor în rezultat este, în ciuda sforţărilor noastre (clauza ORDER BY, folosirea acoladei care are codul ASCII superior oricărei litere), aiurită (vezi figura 11.8).

JUDET	LOC	DENCL	VINZARI
Iasi	Iasi	Client 1 SRL	15061538
Iasi	Iasi	Client 2 SA	361335
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi	15422873
Iasi	Pascani	Client 4	9479110
Iasi	Pascani	{Subtotal-LOCALITATE}Pascani	9479110
Iasi	{Subtotal-JUDET}Iasi		24901983
Neamt	Roman	Client 6 SA	6021707
Neamt	Roman	{Subtotal-LOCALITATE}Roman	6021707
Neamt	{Subtotal-JUDET}Neamt		6021707
Timis	{Subtotal-JUDET}Timis		832812
Timis	Timisoara	Client 5 SRL	432400
Timis	Timisoara	Client 7 SRL	400412
Timis	Timisoara	{Subtotal-LOCALITATE}Timisoara	832812
{TOTAL GENERAL}			49170335
Vaslui	{Subtotal-JUDET}Vaslui		17413835
Vaslui	Vaslui	Client 3 SRL	17413835
Vaslui	Vaslui	{Subtotal-LOCALITATE}Vaslui	17413835

Figura 11.8. (Alte) depresii cauzate de Oracle

Pentru a obţine rezultatul corect, cu liniile dispuse în ordine firească nu avem decât să renunţăm, din interogarea de mai sus, la clauza ORDER BY (vezi figura 11.9)! Nu aş recomanda, însă, această soluţie (este foarte posbil ca pe calculatoarele dumneavoastră ordinea să fie alta), ci inserarea unui spaţiu (cum am procedat cu DataFact în Oracle la problema precedentă) în structura CASE pentru afişarea valorilor coloanei Judeţ:

```
SELECT
        CASE WHEN Judet IS NULL THEN '{TOTAL GENERAL}'
                ELSE ' ' || Judet
        END AS Judet,
        CASE    WHEN Judet IS NULL THEN ''
                ELSE COALESCE(Loc, '{Subtotal-JUDET}' || Judet)
        END AS Loc,
        CASE    WHEN Loc IS NULL THEN ''
```

```
        ELSE COALESCE(DenCl, '{Subtotal-LOCALITATE}' | | Loc)
    END AS DenCl,
    ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM
    – ... identic interogării Oracle precedente
```

JUDET	LOC	DENCL	VINZARI
Iasi	Iasi	Client 1 SRL	15061538
Iasi	Iasi	Client 2 SA	361335
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi	15422873
Iasi	Pascani	Client 4	9479110
Iasi	Pascani	{Subtotal-LOCALITATE}Pascani	9479110
Iasi	{Subtotal-JUDET}Iasi		24901983
Neamt	Roman	Client 6 SA	6021707
Neamt	Roman	{Subtotal-LOCALITATE}Roman	6021707
Neamt	{Subtotal-JUDET}Neamt		6021707
Timis	Timisoara	Client 5 SRL	432400
Timis	Timisoara	Client 7 SRL	400412
Timis	Timisoara	{Subtotal-LOCALITATE}Timisoara	832812
Timis	{Subtotal-JUDET}Timis		832812
Vaslui	Vaslui	Client 3 SRL	17413835
Vaslui	Vaslui	{Subtotal-LOCALITATE}Vaslui	17413835
Vaslui	{Subtotal-JUDET}Vaslui		17413835
{TOTAL GENERAL}			49170335

Figura 11.9. Ordonare corectă în Oracle la folosirea funcţiei ROLLUP

Soluţia a fost aplicată deja în DB2, aşa că nu mai formulăm şi sintaxa acestei interogări pe dialectul SQL de la IBM.

11.1.2. ROLLUP şi GROUPING

Structura alternativă de tip CASE poate utiliza şi o nouă funcţie introdusă în SQL:1999 - GROUPING. Argumentul acesteia este coloana de grupare, rezultatul furnizat fiind 1, atunci când coloana respectivă este inclusă într-un grup de agregare superior, sau 0 pentru liniile „normale" (din afara subtotalurilor). Pentru edificare, să examinăm rezultatul din figura 11.10, produs de interogarea (sintaxă DB2/Oracle):

```
SELECT Judet, Loc, DenCl,
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari,
        GROUPING (Judet) AS Grup_Judet,
        GROUPING (Loc) AS Grup_Loc,
        GROUPING (DenCl) AS Grup_DenCl
FROM judete j
```

```
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY ROLLUP (Judet, Loc, DenCl)
```

JUDET	LOC	DENCL	VINZARI	GRUP_JUDET	GRUP_LOC	GRUP_DENCL
Iasi	Iasi	Client 2 SA	361335	0	0	0
Iasi	Iasi	Client 1 SRL	15061538	0	0	0
Iasi	Iasi	(null)	15422873	0	0	1
Iasi	Pascani	Client 4	9479109,5	0	0	0
Iasi	Pascani	(null)	9479109,5	0	0	1
Iasi	(null)	(null)	24901982,5	0	1	1
Neamt	Roman	Client 6 SA	6021706,5	0	0	0
Neamt	Roman	(null)	6021706,5	0	0	1
Neamt	(null)	(null)	6021706,5	0	1	1
Timis	Timisoara	Client 5 SRL	432400	0	0	0
Timis	Timisoara	Client 7 SRL	400411,5	0	0	0
Timis	Timisoara	(null)	832811,5	0	0	1
Timis	(null)	(null)	832811,5	0	1	1
Vaslui	Vaslui	Client 3 SRL	17413834,5	0	0	0
Vaslui	Vaslui	(null)	17413834,5	0	0	1
Vaslui	(null)	(null)	17413834,5	0	1	1
(null)	(null)	(null)	49170335	1	1	1

Figura 11.10. Valori furnizate de funcţia GROUPING

Valoarea 1 returnată de funcţia GROUPING pentru un atribut indică un subtotal pentru acel atribut, relativ la atributele din stânga (în ordinea declarată în clauza ROLLUP). Pe baza funcţiei GROUPING, rezultatul din figura 11.9 poate fi obţinut şi astfel:

```
SELECT CASE
            WHEN GROUPING (Judet) = 1 THEN '{TOTAL GENERAL}'
            ELSE '' || Judet
       END AS Judet,
       CASE
            WHEN GROUPING (Judet) = 1 THEN ''
            WHEN GROUPING(Loc) = 1 THEN '{Subtotal-JUDET}' || Judet
            ELSE '' || Loc
       END AS Loc,
       CASE
            WHEN GROUPING (Judet) = 1 OR GROUPING(Loc) = 1 THEN ''
```

```
            WHEN GROUPING (DenCl) = 1 THEN '{Subtotal-LOCALITATE}' || Loc
            ELSE ' ' || DenCl
        END AS DenCl,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY ROLLUP (Judet, Loc, DenCl)
ORDER BY 1,2,3
```

Interogarea funcţionează fără modificări în DB2. Schimbând operatorul de concatenare (din || în +) şi clauza GROUP BY din *GROUP BY ROLLUP (Judet, Loc, DenCl)* în *GROUP BY Judet, Loc, DenCl WITH ROLLUP*, interogarea funcţionează şi în MS SQL Server.

11.1.3. ROLLUP-uri parţiale

Este de dorit, uneori, ca subtotalizarea atributelor de grupare să fie parţială: fie nu interesează totalul general, fie subtotalizarea este necesară numai pentru anumite atribute sau grupuri de atribute. Schimbăm cerinţele de raportare, în sensul că acum *vânzările din luna septembrie 2007 ne interesează pe facturi, clienţi, localităţi şi judeţe. Subtotalurile, însă, trebuie calculate numai la nivel de client şi localitate, iar totalul general nu ne priveşte.* Interogarea Oracle ce furnizează răspunsul (figura 11.11) este:

```
SELECT CASE
            WHEN GROUPING (Judet) = 1 THEN '{TOTAL GENERAL}'
            ELSE Judet
        END AS Judet,
        CASE
            WHEN GROUPING (Judet) = 1 THEN ' '
            WHEN GROUPING (Loc) = 1 THEN '{Subtotal-JUDET}' || Judet
            ELSE Loc
        END AS Loc,
        CASE
            WHEN GROUPING (Judet) = 1 OR GROUPING(Loc) = 1 THEN ' '
            WHEN GROUPING (DenCl) = 1 THEN '{Subtotal-LOCALITATE}' || Loc
        ELSE ' ' || DenCl
        END AS DenCl,
        CASE
            WHEN GROUPING (Judet) = 1 OR GROUPING(Loc) = 1
```

```
                  OR GROUPING (DenCl) = 1 THEN ''
              WHEN GROUPING (f.NrFact) = 1 THEN '{Subtotal-CLIENT}' || DenCl
    ELSE '' || CAST(f.NrFact AS CHAR(8)) || ''
    END AS Factura,
    ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
    INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
    INNER JOIN clienti c ON cp.CodPost = c.CodPost
    INNER JOIN facturi f ON c.CodCl = f.CodCl
    INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
    INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
              EXTRACT (MONTH FROM DataFact) = 9
GROUP BY Judet, Loc, ROLLUP (DenCl, f.NrFact)
ORDER BY 1,2,3,4
```

JUDET	LOC	DENCL	FACTURA	VINZARI
Iasi	Iasi	Client 1 SRL	3111	4442960
Iasi	Iasi	Client 1 SRL	3115	110908
Iasi	Iasi	Client 1 SRL	3117	287855
Iasi	Iasi	Client 1 SRL	3118	179565
Iasi	Iasi	Client 1 SRL	{Subtotal-CLIENT}Client 1 SRL	5021287
Iasi	Iasi	Client 2 SA	3113	127530
Iasi	Iasi	Client 2 SA	{Subtotal-CLIENT}Client 2 SA	127530
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		5148817
Timis	Timisoara	Client 5 SRL	3112	153442
Timis	Timisoara	Client 5 SRL	{Subtotal-CLIENT}Client 5 SRL	153442
Timis	Timisoara	Client 7 SRL	3116	136850
Timis	Timisoara	Client 7 SRL	{Subtotal-CLIENT}Client 7 SRL	136850
Timis	Timisoara	{Subtotal-LOCALITATE}Timisoara		290292
Vaslui	Vaslui	Client 3 SRL	3119	5819668
Vaslui	Vaslui	Client 3 SRL	{Subtotal-CLIENT}Client 3 SRL	5819668
Vaslui	Vaslui	{Subtotal-LOCALITATE}Vaslui		5819668

Figura 11.11. ROLLUP parţial

Înlcuind funcţiile EXTRACT cu YEAR şi MONTH se obţine varianta funcţionabilă în DB2. Prin clauza *GROUP BY Judet, Loc, ROLLUP (DenCl, F.NrFact)* se vor calcula două tipuri de subtotaluri:

- unul pentru valori distincte ale combinaţiei (Judeţ, Loc şi DenCl) şi
- altul pentru valori distincte ale combinaţiei (Judeţ, Loc).

Reținem așadar: o coloană declarată în ROLLUP determină calcularea subtotalurilor pentru valori distincte ale celorlalte coloane. Astfel, interogarea Oracle următoare produce raportul din figura 11.12.

```
SELECT
        CASE
                WHEN GROUPING (Judet) = 1 THEN '{TOTAL GENERAL}'
        ELSE Judet
        END AS Judet,
        CASE
                WHEN GROUPING (Judet) = 1 THEN ' '
                WHEN GROUPING(Loc) = 1 THEN '{Subtotal-JUDET}' || Judet
        ELSE Loc
        END AS Loc,
        CASE
                WHEN GROUPING (Judet) = 1 OR GROUPING(Loc) = 1 THEN ' '
                WHEN GROUPING (DenCl) = 1 THEN '{Subtotal-LOCALITATE}' || Loc
        ELSE DenCl
        END AS DenCl,
        CASE
                WHEN GROUPING (Judet) = 1 OR GROUPING(Loc) = 1
                        OR GROUPING (DenCl) = 1 THEN ' '
                WHEN GROUPING (f.NrFact) = 1 THEN '{Subtotal-CLIENT}' || DenCl
        ELSE CAST (f.NrFact AS CHAR(8)) || ' '
        END AS Factura,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007
                AND EXTRACT (MONTH FROM DataFact) = 9
GROUP BY Judet, ROLLUP(Loc, DenCl), f.NrFact
ORDER BY 1,2,3,4
```

Rezultatul este destul de anapoda, dar nici interogarea nu-i mai prejos! Practic, după fiecare valoare distinctă a combinației atributelor din GROUP BY se calculează un subtotal pe județ-localitate și altul pe județ (deoarece atributele din ROLLUP sunt DenCl și Loc).

În SQL Server, ROLLUP-urile parţiale nu sunt posibile direct, ci doar prin grupări clasice şi reuniuni, aşa că nu mai consumăm timp şi spaţiu cu redactarea interogării ce are ca rezultat figura 11.12.

JUDET	LOC	DENCL	FACTURA	VINZARI
Iasi	Iasi	Client 1 SRL	3111	4442960
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		4442960
Iasi	{Subtotal-JUDET}Iasi			4442960
Iasi	Iasi	Client 2 SA	3113	127530
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		127530
Iasi	{Subtotal-JUDET}Iasi			127530
Iasi	Iasi	Client 1 SRL	3115	110908
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		110908
Iasi	{Subtotal-JUDET}Iasi			110908
Iasi	Iasi	Client 1 SRL	3117	287855
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		287855
Iasi	{Subtotal-JUDET}Iasi			287855
Iasi	Iasi	Client 1 SRL	3118	179565
Iasi	Iasi	{Subtotal-LOCALITATE}Iasi		179565
Iasi	{Subtotal-JUDET}Iasi			179565
Timis	Timisoara	Client 5 SRL	3112	153442
Timis	Timisoara	{Subtotal-LOCALITATE}Timisoara		153442
Timis	{Subtotal-JUDET}Timis			153442
Timis	Timisoara	Client 7 SRL	3116	136850
Timis	Timisoara	{Subtotal-LOCALITATE}Timisoara		136850
Timis	{Subtotal-JUDET}Timis			136850
Vaslui	Vaslui	Client 3 SRL	3119	5819668
Vaslui	Vaslui	{Subtotal-LOCALITATE}Vaslui		5819668
Vaslui	{Subtotal-JUDET}Vaslui			5819668

Figura 11.12. Un alt ROLLUP parţial, dar ceva mai curios

11.2. Analize multidimensionale. Operatorii CUBE şi GROUPING SETS

De multe ori, pentru a produce o impresie mai puternică partenerilor de discuţii, analiza datelor trebuie să fie una multidimensională. Apanaj, prin excelenţă, al... Excel-ului sau al pretenţioaselor instrumente OLAP, acest paragraf este menit să demonstreze că prin SQL:1999 sunt create condiţii ideale pentru acest gen de operaţiuni.

11.2.1. Cuburi & cubulețe

Pentru început, ne interesează vizualizarea vânzărilor din luna septembrie 2007 pe două axe, *produse* şi *clienți*, ca în figura 11.13.

CLIENT	PRODUS	VINZARI
Client 1 SRL	Produs 1	324989
Client 1 SRL	Produs 2	411584
Client 1 SRL	Produs 5	4284714
Client 2 SA	Produs 2	127530
Client 3 SRL	Produs 2	41584
Client 3 SRL	Produs 3	33320
Client 3 SRL	Produs 4	84530
Client 3 SRL	Produs 5	5660235
Client 5 SRL	Produs 2	95430
Client 5 SRL	Produs 3	58013
Client 7 SRL	Produs 2	136850
Client 1 SRL	{}	5021287
Client 2 SA	{}	127530
Client 3 SRL	{}	5819668
Client 5 SRL	{}	153442
Client 7 SRL	{}	136850
{}	Produs 1	324989
{}	Produs 2	812977
{}	Produs 3	91333
{}	Produs 4	84530
{}	Produs 5	9944949
{}	{}	11258777

Figura 11.13. Analiză pe două dimensiuni

Întrucât *produse* şi *clienți* sunt două „variabile" independente, vor exista patru variante de grupare a datelor:

- grupare după client şi produs;
- grupare numai după client;
- grupare numai după produs;
- un grup pentru total general.

Soluțiile redactate urmând sintaxa SQL-92 sunt impresionante ca întindere, iar dacă analiza se realizează după trei, patru... variabile, impresionantul se transformă în halucinant. Iată prima interogare Oracle ce obține un raport de genul celui din figura 11.13.

```
SELECT ''|| DenCl AS Client, '' || DenPr AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
```

```
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, DenPr
UNION
SELECT DenCl AS Client,  '{}' AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)  AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl
UNION
SELECT '{}' AS Client, DenPr AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenPr
UNION
SELECT '{}' AS Client,  '{}' AS Produs,
ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)  AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
```

Pentru astfel de situații există un operator cuceritor prin simplitate: CUBE. Cu ajutorul acestuia, se poate redacta următoarea variantă care produce raportul din figura 11.14:

```
SELECT DenCl AS Client, DenPr AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
```

```
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                        EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr )
ORDER BY DenCl, DenPr
```

CLIENT	PRODUS	VINZARI
Client 1 SRL	Produs 1	324989
Client 1 SRL	Produs 2	411584
Client 1 SRL	Produs 5	4284714
Client 1 SRL	(null)	5021287
Client 2 SA	Produs 2	127530
Client 2 SA	(null)	127530
Client 3 SRL	Produs 2	41584
Client 3 SRL	Produs 3	33320
Client 3 SRL	Produs 4	84530
Client 3 SRL	Produs 5	5660235
Client 3 SRL	(null)	5819668
Client 5 SRL	Produs 2	95430
Client 5 SRL	Produs 3	58013
Client 5 SRL	(null)	153442
Client 7 SRL	Produs 2	136850
Client 7 SRL	(null)	136850
(null)	Produs 1	324989
(null)	Produs 2	812977
(null)	Produs 3	91333
(null)	Produs 4	84530
(null)	Produs 5	9944949
(null)	(null)	11258777

Figura 11.14. Un cub neprelucrat

În DB2 trebuie înlocuite cele două funcţii EXTRACT, iar sintaxa MS SQL Server este similară:

```
SELECT DenCl AS Client, DenPr AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DATEPART(YEAR,DataFact) = 2007 AND DATEPART(MONTH,DataFact) = 9
```

```
GROUP BY DenCl, DenPr WITH CUBE
```

CLIENT	PRODUS	VINZARI
Client 1 SRL	Produs 1	324989
Client 1 SRL	Produs 2	411584
Client 1 SRL	Produs 5	4284714
Client 1 SRL	Subtotal client	5021287
Client 2 SA	Produs 2	127530
Client 2 SA	Subtotal client	127530
Client 3 SRL	Produs 2	41584
Client 3 SRL	Produs 3	33320
Client 3 SRL	Produs 4	84530
Client 3 SRL	Produs 5	5660235
Client 3 SRL	Subtotal client	5819668
Client 5 SRL	Produs 2	95430
Client 5 SRL	Produs 3	58013
Client 5 SRL	Subtotal client	153442
Client 7 SRL	Produs 2	136850
Client 7 SRL	Subtotal client	136850
Subtotal produs	Produs 1	324989
Subtotal produs	Produs 2	812977
Subtotal produs	Produs 3	91333
Subtotal produs	Produs 4	84530
Subtotal produs	Produs 5	9944949
Total general		11258777

Figura 11.15. Un cub ceva mai lustruit

Pentru un plus de atractivitate, se poate recurge la clauza GROUPING astfel:

```
SELECT CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 THEN 'Total general'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0
                                THEN 'Subtotal produs '
        ELSE DenCl
        END AS Client,
        CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 THEN ' '
        WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1
        THEN 'Subtotal client '
        ELSE DenPr
        END AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
```

```
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr )
ORDER BY DenCl, DenPr
```

Drept răsplată, forma raportului este cea din figura 11.15. Se mai cuvine de adăugat că atributele care definesc dimensiunile de analiză trebuie să fie de acelaşi tip. Interogarea funcţionează în DB2 (aveţi grijă la EXTRACT) şi în SQL Server, schimbând funcţiile EXTRACT cu DATEPART şi clauza GROUP BY.

Un mare avantaj al CUBE-ului iese la iveală când suntem în criză de idei sau rapoartele sunt prea „subţiri". Trecerea de la analiza pe două axe la cea pe trei axe: clienţi, produse şi zile, este cât se poate de spumoasă:

```
SELECT
        CASE
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1
                        AND GROUPING(DataFact) = 1 THEN '{{{Total general}}}'
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                        GROUPING(DataFact) = 0 THEN '{{Subtotal pentru ziua}}'
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                        GROUPING(DataFact) = 1 THEN '{{Subtotal pentru produs}}'
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                        GROUPING(DataFact) = 0 THEN '{Subtotal produs/zi}'
                ELSE DenCl
        END AS Client,
        CASE
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                        GROUPING(DataFact) = 1 THEN '{{{}}}'
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                        GROUPING(DataFact) = 0 THEN '{{}}'
                WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
                        GROUPING(DataFact) = 0 THEN '{Subtotal client/zi}'
                WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
                        GROUPING(DataFact) = 1 THEN '{{Subtotal client}}'
                ELSE '' || DenPr
        END AS Produs,
        CASE
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
```

```
                    GROUPING(DataFact) = 1 THEN '{{{}}}'
                WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                    GROUPING(DataFact) = 1 THEN '{{}}'
                WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 0 AND
                    GROUPING(DataFact) = 1 THEN '{Subtotal client/produs}'
                ELSE '' ||DataFact || ''
        END AS Zi,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr, DataFact )
ORDER BY 1,2,3
```

Fiind mai lung de o pagină, rezultatul este afişat sub formă de tabel (tabelul 11.1). Atenţie, este posibil să aveţi probleme la ordonarea rezultatului în Oracle.

Tabelul 11.1 Analiză pe trei dimensiuni

CLIENT	PRODUS	ZI	VÂNZĂRI
Client 1 SRL	Produs 1	01-09-2007	67830
Client 1 SRL	Produs 1	10-09-2007	124355
Client 1 SRL	Produs 1	17-09-2007	132804
Client 1 SRL	Produs 1	{Subtotal client/produs}	324989
Client 1 SRL	Produs 2	01-09-2007	90416
Client 1 SRL	Produs 2	02-09-2007	110908
Client 1 SRL	Produs 2	10-09-2007	163500
Client 1 SRL	Produs 2	17-09-2007	46761
Client 1 SRL	Produs 2	{Subtotal client/produs}	411584
Client 1 SRL	Produs 5	01-09-2007	4284714
Client 1 SRL	Produs 5	{Subtotal client/produs}	4284714
Client 1 SRL	{{Subtotal client}}		5021287
Client 1 SRL	{Subtotal client/zi}	01-09-2007	4442960
Client 1 SRL	{Subtotal client/zi}	02-09-2007	110908
Client 1 SRL	{Subtotal client/zi}	10-09-2007	287855
Client 1 SRL	{Subtotal client/zi}	17-09-2007	179565
Client 2 SA	Produs 2	02-09-2007	127530
Client 2 SA	Produs 2	{Subtotal client/produs}	127530
Client 2 SA	{{Subtotal client}}		127530
Client 2 SA	{Subtotal client/zi}	02-09-2007	127530

Client 3 SRL	Produs 2	07-09-2007	41584
Client 3 SRL	Produs 2	{Subtotal client/produs}	41584
Client 3 SRL	Produs 3	07-09-2007	33320
Client 3 SRL	Produs 3	{Subtotal client/produs}	33320
Client 3 SRL	Produs 4	07-09-2007	84530
Client 3 SRL	Produs 4	{Subtotal client/produs}	84530
Client 3 SRL	Produs 5	07-09-2007	5660235
Client 3 SRL	Produs 5	{Subtotal client/produs}	5660235
Client 3 SRL	{{Subtotal client}}		5819668
Client 3 SRL	{Subtotal client/zi}	07-09-2007	5819668
Client 5 SRL	Produs 2	01-09-2007	95430
Client 5 SRL	Produs 2	{Subtotal client/produs}	95430
Client 5 SRL	Produs 3	01-09-2007	58013
Client 5 SRL	Produs 3	{Subtotal client/produs}	58013
Client 5 SRL	{{Subtotal client}}		153442
Client 5 SRL	{Subtotal client/zi}	01-09-2007	153442
Client 7 SRL	Produs 2	10-09-2007	136850
Client 7 SRL	Produs 2	{Subtotal client/produs}	136850
Client 7 SRL	{{Subtotal client}}		136850
Client 7 SRL	{Subtotal client/zi}	10-09-2007	136850
{{Subtotal pentru produs}}	Produs 1	{{}}	324989
{{Subtotal pentru produs}}	Produs 2	{{}}	812977
{{Subtotal pentru produs}}	Produs 3	{{}}	91333
{{Subtotal pentru produs}}	Produs 4	{{}}	84530
{{Subtotal pentru produs}}	Produs 5	{{}}	9944949
{{Subtotal pentru ziua}}	{{}}	01-09-2007	4596402
{{Subtotal pentru ziua}}	{{}}	02-09-2007	238438
{{Subtotal pentru ziua}}	{{}}	07-09-2007	5819668
{{Subtotal pentru ziua}}	{{}}	10-09-2007	424705
{{Subtotal pentru ziua}}	{{}}	17-09-2007	179565
{Subtotal produs/zi}	Produs 1	01-09-2007	67830
{Subtotal produs/zi}	Produs 1	10-09-2007	124355
{Subtotal produs/zi}	Produs 1	17-09-2007	132804
{Subtotal produs/zi}	Produs 2	01-09-2007	185845
{Subtotal produs/zi}	Produs 2	02-09-2007	238438
{Subtotal produs/zi}	Produs 2	07-09-2007	41584
{Subtotal produs/zi}	Produs 2	10-09-2007	300350
{Subtotal produs/zi}	Produs 2	17-09-2007	46761
{Subtotal produs/zi}	Produs 3	01-09-2007	58013

{Subtotal produs/zi}	Produs 3	07-09-2007	33320
{Subtotal produs/zi}	Produs 4	07-09-2007	84530
{Subtotal produs/zi}	Produs 5	01-09-2007	4284714
{Subtotal produs/zi}	Produs 5	07-09-2007	5660235
{{{Total general}}}	{{{}}}	{{{}}}	11258777

DB2 reclamă funcţiile YEAR şi MONTH în locul EXTRACT şi o funcţie CAST necesară conversiei datei facturii în şir de caractere. Pentru a porta soluţia în SQL Server, în afara funcţiei DATEPART şi sintaxei GROUP BY... WITH CUBE, este necesară, ca în DB2, funcţia CAST:

```
SELECT
    CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1
            AND GROUPING(DataFact) = 1 THEN '{{{Total general}}}'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 0 THEN '{{Subtotal pentru ziua}}'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
            GROUPING(DataFact) = 1 THEN '{{Subtotal pentru produs}}'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
            GROUPING(DataFact) = 0 THEN '{Subtotal produs/zi}'
        ELSE DenCl
    END AS Client,
    CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 1 THEN '{{{}}}'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 0 THEN '{{}}'
        WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 0 THEN '{Subtotal client/zi}'
        WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 1 THEN '{{Subtotal client}}'
        ELSE ' ' + DenPr
    END AS Produs,
    CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
            GROUPING(DataFact) = 1 THEN '{{{}}}'
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
            GROUPING(DataFact) = 1 THEN '{{}}'
        WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 0 AND
            GROUPING(DataFact) = 1 THEN '{Subtotal client/produs}'
```

```
            ELSE ' ' + CAST (DataFact AS VARCHAR(12)) + ' '
        END AS Zi,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE DATEPART(YEAR,DataFact) = 2007 AND DATEPART(MONTH,DataFact) = 9
GROUP BY DenCl, DenPr, DataFact WITH CUBE
```

11.2.2. Operatorul GROUPING SETS

Uneori, prea multe detalii, precum cele din tabelul de mai sus, devin obositoare. Prin GROUPING SETS se poate regla granularitatea agregărilor. Dacă, spre exemplu, în interogarea anterioară se doreşte urmărirea vânzărilor zilnice pentru fiecare client (şi toate produsele) şi pentru fiecare produs (şi toţi clienţii), se poate recurge la varianta următoare:

CLIENT	PRODUS	ZI	VINZARI
Client 1 SRL	Produs 1	{Subtotal client/produs}	324989
Client 1 SRL	Produs 2	{Subtotal client/produs}	411584
Client 1 SRL	Produs 5	{Subtotal client/produs}	4284714
Client 2 SA	Produs 2	{Subtotal client/produs}	127530
Client 3 SRL	Produs 2	{Subtotal client/produs}	41584
Client 3 SRL	Produs 3	{Subtotal client/produs}	33320
Client 3 SRL	Produs 4	{Subtotal client/produs}	84530
Client 3 SRL	Produs 5	{Subtotal client/produs}	5660235
Client 5 SRL	Produs 2	{Subtotal client/produs}	95430
Client 5 SRL	Produs 3	{Subtotal client/produs}	58013
Client 7 SRL	Produs 2	{Subtotal client/produs}	136850
{{Subtotal pentru ziua}}	{{}}	01-09-2007	4596402
{{Subtotal pentru ziua}}	{{}}	02-09-2007	238438
{{Subtotal pentru ziua}}	{{}}	07-09-2007	5819668
{{Subtotal pentru ziua}}	{{}}	10-09-2007	424705
{{Subtotal pentru ziua}}	{{}}	17-09-2007	179565

Figura 11.16. Clauza GROUPING SETS

```
SELECT      ...
FROM        ... – identic interogării anterioare
WHERE       ...
GROUP BY GROUPING SETS ( (DenCl, DenPr), DataFact)
ORDER BY 1,2,3
```

În clauza GROUP BY a fost introdus operatorul GROUPING SETS prin care se precizează două grupuri de agregare, unul alcătuit din combinaţia (DenCl, DenPr), celălalt din data facturii, astfel încât se obţine un rezultat precum cel din figura 11.16.

Cu modificările amintite în exemplele precedente, interogarea funcţionează şi în DB2. Dacă se doreşte o situaţie sintetică cu subtotaluri ale celor trei atribute plus un total general (obţinut prin perechea de paranteze fără argumente) se schimbă clauza GROUP BY astfel:

```
SELECT      ...
FROM        ... – identic interogării anterioare
WHERE       ...
GROUP BY GROUPING SETS (DenCl, DenPr, DataFact, ())
ORDER BY 1,2,3
```

CLIENT	PRODUS	ZI	VINZARI
Client 1 SRL	{{Subtotal client}}		5021287
Client 2 SA	{{Subtotal client}}		127530
Client 3 SRL	{{Subtotal client}}		5819668
Client 5 SRL	{{Subtotal client}}		153442
Client 7 SRL	{{Subtotal client}}		136850
{{Subtotal pentru produs}}	Produs 1	{()}	324989
{{Subtotal pentru produs}}	Produs 2	{()}	812977
{{Subtotal pentru produs}}	Produs 3	{()}	91333
{{Subtotal pentru produs}}	Produs 4	{()}	84530
{{Subtotal pentru produs}}	Produs 5	{()}	9944949
{{Subtotal pentru ziua}}	{()}	01-09-2007	4596402
{{Subtotal pentru ziua}}	{()}	02-09-2007	238438
{{Subtotal pentru ziua}}	{()}	07-09-2007	5819668
{{Subtotal pentru ziua}}	{()}	10-09-2007	424705
{{Subtotal pentru ziua}}	{()}	17-09-2007	179565
{{{Total general}}}	{{()}}	{{()}}	11258777

Figura 11.17. GROUPING SETS – exemplul 2

Cele 16 linii din figura 11.17 sunt obţinute astfel: 5 clienţi pentru care sunt vânzări în septembrie 2007, plus 5 produse vândute, plus 5 zile în care s-au întocmit facturi, plus linia totalului general.

Interesant este şi faptul că cei trei operatori pot fi combinaţi. Spre exemplu, dacă GROUP BY are forma:

```
...
GROUP BY
GROUPING SETS ( ROLLUP (DenCl, DenPr), DataFact)
...
```

prin execuţia consultării se obţin liniile din figura 11.18.

SQL Server nu are, după bruma noastră de cunoştinţe, opţiunea GROUPING SETS.

CLIENT	PRODUS	ZI	VINZARI
Client 1 SRL	Produs 1	{Subtotal client/produs}	324989
Client 1 SRL	Produs 2	{Subtotal client/produs}	411584
Client 1 SRL	Produs 5	{Subtotal client/produs}	4284714
Client 1 SRL	{{Subtotal client}}		5021287
Client 2 SA	Produs 2	{Subtotal client/produs}	127530
Client 2 SA	{{Subtotal client}}		127530
Client 3 SRL	Produs 2	{Subtotal client/produs}	41584
Client 3 SRL	Produs 3	{Subtotal client/produs}	33320
Client 3 SRL	Produs 4	{Subtotal client/produs}	84530
Client 3 SRL	Produs 5	{Subtotal client/produs}	5660235
Client 3 SRL	{{Subtotal client}}		5819668
Client 5 SRL	Produs 2	{Subtotal client/produs}	95430
Client 5 SRL	Produs 3	{Subtotal client/produs}	58013
Client 5 SRL	{{Subtotal client}}		153442
Client 7 SRL	Produs 2	{Subtotal client/produs}	136850
Client 7 SRL	{{Subtotal client}}		136850
{{Subtotal pentru ziua}}	{{}}	01-09-2007	4596402
{{Subtotal pentru ziua}}	{{}}	02-09-2007	238438
{{Subtotal pentru ziua}}	{{}}	07-09-2007	5819668
{{Subtotal pentru ziua}}	{{}}	10-09-2007	424705
{{Subtotal pentru ziua}}	{{}}	17-09-2007	179565
{{{Total general}}}	{{{}}}	{{{}}}	11258777

Figura 11.18. GROUPING SETS şi ROLLUP

11.2.3. CUBE-uri parţiale

Discuţia se poartă în termeni similari ROLLUP-ului parţial. Subtotalurile şi combinaţiile posibile sunt limitate la atributele-dimensiuni incluse între paranteze. Spre exemplu, dacă GROUP BY are forma:

GROUP BY atribut1, CUBE (atribut2, atribut3)

vor fi calculate patru subtotaluri, pentru combinaţiile:

- (atribut1, atribut2, atribut3) - GROUP BY-ul obişnuit;
- (atribut1, atribut2);
- (atribut1, atribut3);
- (atribut1).

Astfel, interogarea:

```
SELECT CASE
        WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1
            AND GROUPING(DataFact) = 1 THEN '{{{Total general}}}'
```

```
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1
                  AND GROUPING(DataFact) = 0 THEN '{{Subtotal pentru
            ziua}}'
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                  GROUPING(DataFact) = 1 THEN '{{Subtotal pentru produs}}'
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                  GROUPING(DataFact) = 0 THEN '{Subtotal produs/zi}'
            ELSE DenCl
      END AS Client,
      CASE
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                  GROUPING(DataFact) = 1 THEN '{{{}}}'
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                  GROUPING(DataFact) = 0 THEN '{{}}'
            WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
                  GROUPING(DataFact) = 0 THEN '{Subtotal client/zi}'
            WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 1 AND
                  GROUPING(DataFact) = 1 THEN '{{Subtotal client}}'
            ELSE ' ' || DenPr
      END AS Produs,
      CASE
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 1 AND
                  GROUPING(DataFact) = 1 THEN '{{{}}}'
            WHEN GROUPING(DenCl) = 1 AND GROUPING(DenPr) = 0 AND
                  GROUPING(DataFact) = 1 THEN '{{}}'
            WHEN GROUPING(DenCl) = 0 AND GROUPING(DenPr) = 0 AND
                  GROUPING(DataFact) = 1 THEN '{Subtotal client/produs}'
            ELSE ' ' ||CAST (DataFact AS CHAR(10)) || ' '
      END AS Zi,
      ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
FROM clienti c
      INNER JOIN facturi f ON c.CodCl = f.CodCl
      INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
      INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                        EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DenCl, CUBE  (DenPr, DataFact )
ORDER BY 1,2,3
```

va genera liniile tabelului 11.2.

Tabel 11.2. CUBE parţial

CLIENT	PRODUS	ZI	VÂNZĂRI
Client 1 SRL	Produs 1	01-09-2007	67830
Client 1 SRL	Produs 1	10-09-2007	124355
Client 1 SRL	Produs 1	17-09-2007	132804
Client 1 SRL	Produs 1	{Subtotal client/produs}	324989
Client 1 SRL	Produs 2	01-09-2007	90416
Client 1 SRL	Produs 2	02-09-2007	110908
Client 1 SRL	Produs 2	10-09-2007	163500
Client 1 SRL	Produs 2	17-09-2007	46761
Client 1 SRL	Produs 2	{Subtotal client/produs}	411584
Client 1 SRL	Produs 5	01-09-2007	4284714
Client 1 SRL	Produs 5	{Subtotal client/produs}	4284714
Client 1 SRL	{{Subtotal client}}		5021287
Client 1 SRL	{Subtotal client/zi}	01-09-2007	4442960
Client 1 SRL	{Subtotal client/zi}	02-09-2007	110908
Client 1 SRL	{Subtotal client/zi}	10-09-2007	287855
Client 1 SRL	{Subtotal client/zi}	17-09-2007	179565
Client 2 SA	Produs 2	02-09-2007	127530
Client 2 SA	Produs 2	{Subtotal client/produs}	127530
Client 2 SA	{{Subtotal client}}		127530
Client 2 SA	{Subtotal client/zi}	02-09-2007	127530
Client 3 SRL	Produs 2	07-09-2007	41584
Client 3 SRL	Produs 2	{Subtotal client/produs}	41584
Client 3 SRL	Produs 3	07-09-2007	33320
Client 3 SRL	Produs 3	{Subtotal client/produs}	33320
Client 3 SRL	Produs 4	07-09-2007	84530
Client 3 SRL	Produs 4	{Subtotal client/produs}	84530
Client 3 SRL	Produs 5	07-09-2007	5660235
Client 3 SRL	Produs 5	{Subtotal client/produs}	5660235
Client 3 SRL	{{Subtotal client}}		5819668
Client 3 SRL	{Subtotal client/zi}	07-09-2007	5819668
Client 5 SRL	Produs 2	01-09-2007	95430
Client 5 SRL	Produs 2	{Subtotal client/produs}	95430
Client 5 SRL	Produs 3	01-09-2007	58013
Client 5 SRL	Produs 3	{Subtotal client/produs}	58013
Client 5 SRL	{{Subtotal client}}		153442
Client 5 SRL	{Subtotal client/zi}	01-09-2007	153442
Client 7 SRL	Produs 2	10-09-2007	136850
Client 7 SRL	Produs 2	{Subtotal client/produs}	136850
Client 7 SRL	{{Subtotal client}}		136850
Client 7 SRL	{Subtotal client/zi}	10-09-2007	136850

În schimb, dacă GROUP BY are forma:

```
...
GROUP BY CUBE((DenCl, DenPr), DataFact)
```

vor fi calculate patru subtotaluri (tabelul 11.3), pentru combinaţiile:

- (DenCl, DenPr, DataFact) - GROUP BY-ul obişnuit;
- (DenCl, DataFact);
- (DenPr, DataFact);
- (DataFact).

Tabelul 11.3. Alt CUBE parţial

CLIENT	PRODUS	ZI	VÂNZĂRI
Client 1 SRL	Produs 1	01-09-2007	67830
Client 1 SRL	Produs 1	10-09-2007	124355
Client 1 SRL	Produs 1	17-09-2007	132804
Client 1 SRL	Produs 1	{Subtotal client/produs}	324989
Client 1 SRL	Produs 2	01-09-2007	90416
Client 1 SRL	Produs 2	02-09-2007	110908
Client 1 SRL	Produs 2	10-09-2007	163500
Client 1 SRL	Produs 2	17-09-2007	46761
Client 1 SRL	Produs 2	{Subtotal client/produs}	411584
Client 1 SRL	Produs 5	01-09-2007	4284714
Client 1 SRL	Produs 5	{Subtotal client/produs}	4284714
Client 2 SA	Produs 2	02-09-2007	127530
Client 2 SA	Produs 2	{Subtotal client/produs}	127530
Client 3 SRL	Produs 2	07-09-2007	41584
Client 3 SRL	Produs 2	{Subtotal client/produs}	41584
Client 3 SRL	Produs 3	07-09-2007	33320
Client 3 SRL	Produs 3	{Subtotal client/produs}	33320
Client 3 SRL	Produs 4	07-09-2007	84530
Client 3 SRL	Produs 4	{Subtotal client/produs}	84530
Client 3 SRL	Produs 5	07-09-2007	5660235
Client 3 SRL	Produs 5	{Subtotal client/produs}	5660235
Client 5 SRL	Produs 2	01-09-2007	95430
Client 5 SRL	Produs 2	{Subtotal client/produs}	95430
Client 5 SRL	Produs 3	01-09-2007	58013
Client 5 SRL	Produs 3	{Subtotal client/produs}	58013
Client 7 SRL	Produs 2	10-09-2007	136850
Client 7 SRL	Produs 2	{Subtotal client/produs}	136850
{{Subtotal pentru ziua}}	{()}	01-09-2007	4596402
{{Subtotal pentru ziua}}	{()}	02-09-2007	238438
{{Subtotal pentru ziua}}	{()}	07-09-2007	5819668

{{Subtotal pentru ziua}}	{{}}	10-09-2007	424705
{{Subtotal pentru ziua}}	{{}}	17-09-2007	179565
{{{Total general}}}	{{{}}}	{{{}}}	11258777

Nici în materie de cuburi parțiale MS SQL Server nu are vreo clauză/opțiune.

11.3. Clasamente - soluții clasice și OLAP

Problema TOP-urilor a fost atacată (cu oarecare succes) încă din paragraful 9.2, soluțiile fiind subconsultările pe mai multe nivele sau cele proprietare - LIMIT, TOP și FETCH. De asemenea, în paragraful 9.4.2 au fost prezentate câteva soluții proprietare Oracle bazate pe ROWNUM și subconsultări în clauza FROM.

Să se afișeze clasamentul zilelor în ordinea descrescătoare a numărului de facturi emise.

Problema care ne privește acum este indicarea poziției fiecărei zile (vezi figura 11.19). După cum am văzut în paragraful 9.4.2, în Oracle este posibilă ordonarea liniilor unei subconsultări din clauza FROM, așa că soluția următoare, bazată pe pseudocoloana ROWNUM, funcționează:

DATAFACT	NR_FACTURILOR	POZITIE
01-08-2007	4	1
07-08-2007	4	1
14-08-2007	3	3
16-08-2007	2	4
15-08-2007	2	4
01-09-2007	2	4
02-08-2007	2	4
21-08-2007	2	4
02-09-2007	2	4
10-09-2007	2	4
22-08-2007	1	11
03-08-2007	1	11
07-09-2007	1	11
17-09-2007	1	11
04-08-2007	1	11

Figura 11.19. Un clasament veritabil

```
SELECT DataFact, Nr_Facturilor, ROWNUM AS Pozitie
FROM
      (SELECT DataFact, COUNT(*) AS Nr_Facturilor
      FROM facturi
      GROUP BY DataFact
      ORDER BY Nr_Facturilor DESC)
```

Chiar dacă ne putem declara satisfăcuţi de rezultat (figura 11.20), se poate argumenta că 7 august 2007 este pe nedrept „retrogadată" pe locul 2 al clasamentului, atât timp cât are acelaşi număr de zile ca şi 1 august şi nu există nici un criteriu suplimentar de departajare. Interogarea de mai sus este corectă sută la sută doar atunci când, fie toate valorile ordonate sunt diferite, fie când există un număr suficient de criterii de balotaj care să confere unicitate fiecărei poziţii a clasamentului.

DATAFACT	NR_FACTURILOR	POZITIE
01-08-2007	4	1
07-08-2007	4	2
14-08-2007	3	3
16-08-2007	2	4
15-08-2007	2	5
01-09-2007	2	6
02-08-2007	2	7
21-08-2007	2	8
02-09-2007	2	9
10-09-2007	2	10
22-08-2007	1	11
03-08-2007	1	12
07-09-2007	1	13
17-09-2007	1	14
04-08-2007	1	15

Figura 11.20. Pseudoclasamentul obţinut printr-o subconsultare

Există remedii „proprietare" bazate pe expresii-tabelă şi corelări în clauza SELECT. Interogarea Oracle următoare obţine un rezultat precum cel din figura 11.19:

```
WITH top1 AS
        (SELECT DataFact, Nr_Facturilor, ROWNUM AS Pozitie
        FROM
                (SELECT DataFact, COUNT(*) AS Nr_Facturilor
                FROM facturi
                GROUP BY DataFact
                ORDER BY Nr_Facturilor DESC)
        )
SELECT DataFact, Nr_Facturilor,
        CASE WHEN Nr_Facturilor = (SELECT Nr_facturilor
                                FROM top1
                                WHERE pozitie = t2.pozitie - 1
                                )
        THEN (SELECT MIN(Pozitie) FROM top1
        WHERE Nr_facturilor = t2.Nr_Facturilor)
        ELSE Pozitie
```

```
        END AS Pozitie
FROM top1 t2
```

Pentru astfel de situații, *Amendamentul OLAP din SQL:1999* a introdus două funcții ideale, RANK şi DENSE_RANK. În prealabil, însă, câteva chestiuni ce țin de logica execuției unui asemenea gen de consultări. Procesarea interogărilor ce utilizează funcții analitice se derulează în trei etape:

1. se operează toate joncțiunile, se constituie grupurile şi se efectuează selecția asupra grupurilor, altfel spus, se execută clauzele JOIN (sau FROM/WHERE), WHERE, GROUP BY şi HAVING;
2. se „aranjează" rezultatul în vederea aplicării funcțiilor analitice şi se efectuează calculele (sunt create partițiile), apoi
3. funcțiile analitice sunt aplicate linie cu linie în cadrul fiecărei partiții.

Pasul 3 este operațional numai dacă interogarea prezintă la sfârşit o clauză ORDER BY, ceea ce atrage ordonarea finală a rezultatului în conformitate cu criteriile specificate. Partițiile sunt seturi de linii create după delimitarea grupurilor prin GROUP BY, astfel încât pot constitui subiectul (sau obiectul) oricărei funcții de agregare (SUM, AVG...). Constituirea unei partiții se poate face în funcție de valorile unuia sau mai multor atribute sau expresii de atribute.

Pentru a obține o situație identică celei din figura 11.19, soluția comună DB2/Oracle/SQL Server este:

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor,
        RANK() OVER (ORDER BY COUNT(*) DESC) AS Pozitie
FROM facturi
GROUP BY DataFact
```

Funcția RANK ierarhizează după valorile obținute de COUNT(*) liniile rezultatului şi, ceea ce este cel mai important, la valori egale, atribuie fiecărei linii aceeaşi poziție în clasament.

Luăm un alt exemplu care să pună în valoare şi partițiile: *Să se afişeze, pentru fiecare zi de vânzări din luna august 2007, topul celor mai „valoroase" facturi.*

Să analizăm problema:

- partiția are dimensiunea unei zile; numărul liniilor depinde de numărul facturilor întocmite în ziua respectivă; prin urmare, atributul de partiționare este DataFact;
- în cadrul unei partiții, clasamentul se face în funcție de valorile calculate de funcția SUM;
- în rezultatul final, pentru fiecare factură, liniile sunt dispuse în funcție de valorile DataFact şi NrFact, deci, la prezentare, se păstrează ordinea cronologică.

Iată şi soluția DB2/Oracle/MS SQL Server (în DB2 şi SQL Server funcția *EXTRACT* trebuie înlocuită cu *YEAR/MONTH* sau *DATEPART*), rezultatul fiind cel din figura 11.21:

```
SELECT DataFact, F.NrFact,
```

```
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValFact,
RANK() OVER
        (PARTITION BY DataFact
        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
AS Pozitie
FROM produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 8
GROUP BY DataFact, F.NrFact
ORDER BY 1, 4
```

DATAFACT	NRFACT	VALFACT	POZITIE
01-08-2007	1114	6021707	1
01-08-2007	1111	4346038	2
01-08-2007	1112	125516	3
01-08-2007	1113	106275	4
02-08-2007	1115	151238	1
02-08-2007	1116	126713	2
03-08-2007	1117	222050	1
04-08-2007	1118	201975	1
07-08-2007	1119	5774499	1
07-08-2007	1121	4737838	2
07-08-2007	1120	97664	3
14-08-2007	2111	4442960	1
14-08-2007	2112	153442	2
14-08-2007	2113	127530	3
15-08-2007	2116	136850	1
15-08-2007	2115	110908	2
16-08-2007	2117	287855	1
16-08-2007	2118	179565	2
21-08-2007	2119	5819668	1
21-08-2007	2121	4741272	2

Figura 11.21. Topurile zilnice ale facturilor

Factura 1111 este a doua ca valoare pe 1 august, 2007, cea mai mare valoare în această zi având-o factura 1114.

Într-o singură frază SELECT se pot întocmi, pentru aceleaşi date, clasamente diferite prin specificarea mai multor partiţii şi criterii. Astfel, interogarea (Oracle/ DB2/SQL Server)[2]:

2. Sub aceeaşi rezervă a înlocuirii EXTRACT cu YEAR/MONTH sau DATEPART.

```
SELECT DataFact, DenPr,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValFact,
        RANK() OVER (PARTITION BY DataFact
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Pozitie_Zi,
        RANK() OVER (PARTITION BY DenPr
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Pozitie_Prod
FROM produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY DataFact, DenPr
ORDER BY DataFact, DenPr
```

calculează valoarea vânzărilor pe zilele din luna septembrie 2007 (DataFact) pentru fiecare produs. Cele două funcţii RANK determină poziţia vânzărilor pentru un produs şi o zi în totalul vânzărilor pe ziua respectivă (prima) şi poziţia în clasamentul tuturor zilelor în care s-a vândut produsul respectiv (vezi figura 11.22).

DATAFACT	DENPR	VALFACT	POZITIE_ZI	POZITIE_PROD
01-09-2007	Produs 1	67830	3	3
01-09-2007	Produs 2	185845	2	3
01-09-2007	Produs 3	58013	4	1
01-09-2007	Produs 5	4284714	1	2
02-09-2007	Produs 2	238438	1	2
07-09-2007	Produs 2	41584	3	5
07-09-2007	Produs 3	33320	4	2
07-09-2007	Produs 4	84530	2	1
07-09-2007	Produs 5	5660235	1	1
10-09-2007	Produs 1	124355	2	2
10-09-2007	Produs 2	300350	1	1
17-09-2007	Produs 1	132804	1	1
17-09-2007	Produs 2	46761	2	4

Figura 11.22. Două partiţii de ordonare

Pe prima linie a rezultatului apar ziua de 1 septembrie 2007 şi produsul 1, pentru care valoarea vânzărilor în această zi este de 67830 RON. 67830 este, ca mărime, a treia valoare pentru ziua respectivă - *Poziţie_Zi* - (pe primul loc în 1 septembrie este produsul 5, iar al doilea este produsul 2). Între toate zilele în care s-a vândut produsul 1 (1, 10 şi 17 septembrie) - *PoziţieProd* -, valoarea 67830 este tot a treia (pe primul loc sunt vânzările din 17 septembrie, iar pe locul 2 cele din 10 sep-

tembrie). Altfel spus, pentru 1 septembrie 2007, produsul 1 este al treilea în topul pe produse al zilei (cel mai bine vândut în această zi fiind produsul 5), iar dintre toate zilele în care a fost vândut produsul 1, data de 1 septembrie este a treia în topul vânzărilor pe zile (cel mai bine s-a vândut pe 17 septembrie).

Se poate alcătui un clasament nu numai pe baza unei simple funcţii-agregat, ci şi prin combinarea funcţiei RANK cu funcţiile ROLLUP şi CUBE. Rezultatele nu sunt însă întotdeauna clare. Spre exemplu, consultarea DB2/Oracle următoare produce raportul din figura 11.23[3].

```
SELECT Judet, DenPr AS "Produs",
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
        RANK() OVER (PARTITION BY Judet
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS "Poz_in_Judet",
        RANK() OVER (PARTITION BY DenPr
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS "Poz_in_Produs"
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
GROUP BY ROLLUP (Judet, DenPr)
ORDER BY Judet, DenPr
```

Este clar că *Poz_în_Judeţ* va fi întotdeauna 1 în liniile dedicate subtotalului pentru fiecare dintre judeţe. Spre exemplu, produsul 1 nu este al patrulea ca vânzări în judeţul Iaşi, ci al treilea. Valorile *Poz_în_Produs* pentru judeţe sunt însă corecte (nu necesită ajustări). Judeţul în care produsul 4 s-a vândut cel mai bine este Vaslui (245904 RON), iar al doilea clasat este judeţul Neamţ (55754 RON).

Dacă examinăm liniile subtotalurilor pe judeţe şi linia totalului general, aflăm, spre exemplu, din penultima linie, că judeţul Vaslui este al treilea ca volum total al vânzărilor. Poziţia exactă a Vasluiului în topul judeţelor este însă a doua, deoarece prima îi revine automat totalului general (ultima linie).

3. În MS SQL Server interogarea funcţionează cu o uşoară modificare a clauzei GROUP BY (*GROUP BY Judet, DenPr WITH ROLLUP*), iar liniile de total general şi subtotal vor fi plasate la începutul raportului/grupurilor.

JUDET	Produs	VINZARI	Poz_in_Judet	Poz_in_Produs
Iasi	Produs 1	988533	4	1
Iasi	Produs 2	1896546	3	1
Iasi	Produs 5	22016904	2	1
Iasi	(null)	24901983	1	2
Neamt	Produs 2	81641	3	4
Neamt	Produs 4	55754	4	2
Neamt	Produs 5	5884312	2	3
Neamt	(null)	6021707	1	4
Timis	Produs 2	681087	2	2
Timis	Produs 3	151725	3	1
Timis	(null)	832812	1	5
Vaslui	Produs 2	124751	4	3
Vaslui	Produs 3	99960	5	2
Vaslui	Produs 4	245904	3	1
Vaslui	Produs 5	16943220	2	2
Vaslui	(null)	17413835	1	3
(null)	(null)	49170335	1	1

Figura 11.23. RANK şi ROLLUP

Lucrurile stau aidoma şi în cazul operatorului CUBE. Deşi raportul (figura 11.24) legat de vânzările pe clienţi şi produse din luna septembrie 2007, furnizat de următoarea frază SELECT valabilă deopotrivă în Oracle şi DB2 (cu modificările amintite), este bogat în informaţii, câteva dintre acestea trebuie nuanţate.

```
SELECT
        CASE WHEN GROUPING (DenCl) = 1
                THEN '{SUBTOTAL produs}' ELSE DenCl
        END AS Client,
        CASE WHEN GROUPING (DenPr) = 1
                THEN '{SUBTOTAL client}' ELSE DenPr
        END AS Produs,
        SUM(Cantitate * PretUnit * (1 + ProcTVA))  AS Vanzari,
        RANK() OVER (PARTITION BY DenCl
                        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Client,
        RANK() OVER (PARTITION BY DenPr
                        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Produs
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
```

```
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
              EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr)
ORDER BY DenCl, DenPr
```

CLIENT	PRODUS	VINZARI	POZ_CLIENT	POZ_PRODUS
Client 1 SRL	Produs 1	324989	4	1
Client 1 SRL	Produs 2	411584	3	2
Client 1 SRL	Produs 5	4284714	2	3
Client 1 SRL	{SUBTOTAL client}	5021287	1	3
Client 2 SA	Produs 2	127530	1	4
Client 2 SA	{SUBTOTAL client}	127530	1	6
Client 3 SRL	Produs 2	41583,5	4	6
Client 3 SRL	Produs 3	33320	5	3
Client 3 SRL	Produs 4	84529,5	3	1
Client 3 SRL	Produs 5	5660235	2	2
Client 3 SRL	{SUBTOTAL client}	5819668	1	2
Client 5 SRL	Produs 2	95429,5	2	5
Client 5 SRL	Produs 3	58012,5	3	2
Client 5 SRL	{SUBTOTAL client}	153442	1	4
Client 7 SRL	Produs 2	136849,5	1	3
Client 7 SRL	{SUBTOTAL client}	136849,5	1	5
{SUBTOTAL produs}	Produs 1	324989	4	1
{SUBTOTAL produs}	Produs 2	812976,5	3	1
{SUBTOTAL produs}	Produs 3	91332,5	5	1
{SUBTOTAL produs}	Produs 4	84529,5	6	1
{SUBTOTAL produs}	Produs 5	9944949	2	1
{SUBTOTAL produs}	{SUBTOTAL client}	11258776,5	1	1

Figura 11.24. RANK şi CUBE

Astfel, ca vânzări, pentru Client 1 SRL, produsul 5 este pe locul 1 (4284714 RON), şi nu pe locul 2, cum indică a doua linie din figură. La modul general, poziţia produselor în vânzările corespunzătoate unui client (*Poz_Client*) este „umflată" cu 1 din cauza subtotalurilor de la nivel de client. Excepţie fac clienţii cărora li s-a vândut un singur produs (Client 2 SA şi Client 7 SRL), la care ambele valori ale *Poz_Client* sunt 1.

Analog, prima poziţie a listei pentru *Poz_Produs* este cea a subtotalului pe produs, iar, spre exemplu, cel mai bun cumpărător al produsului 2 este Client 1 SRL. Faptul că la produsul 1, în dreptul Clientului 1 SRL este indicată poziţia corectă se datorează vânzării acestui produs unui singur client.

Ţinând seama de interferenţa subtotalurilor, am fi tentaţi să decrementăm rezultatele funcţiilor RANK cu o unitate:

```
SELECT CASE WHEN GROUPING (DenCl) = 1 THEN '{SUBTOTAL produs}'
              ELSE DenCl
        END AS Client,
```

```
        CASE WHEN GROUPING (DenPr) = 1 THEN '{SUBTOTAL client}'
                ELSE DenPr
        END AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
        RANK() OVER (PARTITION BY DenCl
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC) - 1
                AS Poz_Client,
        RANK() OVER (PARTITION BY DenPr
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC) - 1
                AS Poz_Produs
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr )
ORDER BY DenCl, DenPr
```

Rezultatele ar fi perfecte, dacă n-ar exista produse vândute unui singur client (Produs 1) şi clienţi care au cumpărat un singur produs (Client 2 şi Client 7) (vezi figura 11.25). Pentru aceste două speţe, *Poz_Client*, respectiv *Poz_Produs* sunt 0, chiar dacă respectivul produs/client ar fi singur şi poziţia ar trebui să fie, inevitabil, 1. Am modificat, astfel, gluma aceea răsuflată: să alergi de unul singur şi ajungi pe locul zero!

CLIENT	PRODUS	VINZARI	POZ_CLIENT	POZ_PRODUS
Client 1 SRL	Produs 1	324989	3	0
Client 1 SRL	Produs 2	411584	2	1
Client 1 SRL	Produs 5	4284714	1	2
Client 1 SRL	{SUBTOTAL client}	5021287	0	2
Client 2 SA	Produs 2	127530	0	3
Client 2 SA	{SUBTOTAL client}	127530	0	5
Client 3 SRL	Produs 2	41584	3	5
Client 3 SRL	Produs 3	33320	4	2
Client 3 SRL	Produs 4	84530	2	0
Client 3 SRL	Produs 5	5660235	1	1
Client 3 SRL	{SUBTOTAL client}	5819668	0	1
Client 5 SRL	Produs 2	95430	1	4
Client 5 SRL	Produs 3	58013	2	1
Client 5 SRL	{SUBTOTAL client}	153442	0	3
Client 7 SRL	Produs 2	136850	0	2
Client 7 SRL	{SUBTOTAL client}	136850	0	4
{SUBTOTAL produs}	Produs 1	324989	3	0
{SUBTOTAL produs}	Produs 2	812977	2	0
{SUBTOTAL produs}	Produs 3	91333	4	0
{SUBTOTAL produs}	Produs 4	84530	5	0
{SUBTOTAL produs}	Produs 5	9944949	1	0
{SUBTOTAL produs}	{SUBTOTAL client}	11258777	0	0

Figura 11.25. Încercare de corectare a poziţiilor din clasament

O soluţie mai elegantă se bazează pe modificarea modului de declarare a partiţiilor. Pentru a pune în evidenţă cele două modalităţi, se afişează câte două poziţii pentru clienţi/produse, folosindu-se câte două funcţii RANK: una identică cu exemplul anterior şi o alta care beneficiază de serviciile GROUPING, după cum urmează:

```
SELECT
        CASE WHEN GROUPING (DenCl) = 1
                THEN '{SUBTOTAL produs}' ELSE DenCl END AS Client,
        CASE WHEN GROUPING (DenPr) = 1 THEN '{SUBTOTAL client}'
                        ELSE DenPr END AS Produs,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
        RANK() OVER (PARTITION BY DenCl
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Client_1,
        RANK() OVER (PARTITION BY DenCl, GROUPING(DenPr)
                        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Client_2,
        RANK() OVER (PARTITION BY DenPr
```

```
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Produs_1,
        RANK() OVER (PARTITION BY DenPr, GROUPING (DenCl)
                ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
        AS Poz_Produs_2
FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
        INNER JOIN clienti c ON cp.CodPost = c.CodPost
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
GROUP BY CUBE (DenCl, DenPr )
ORDER BY GROUPING(DenCl), DenCl, GROUPING(DenPr), DenPr
```

Coloanele *Poz_Client_2* şi *Poz_Produs_2* conţin, graţie includerii clauzelor GROUPING în funcţiile RANK, valorile care ne interesează (vezi figura 11.26).

CLIENT	PRODUS	VINZARI	POZ_CLIENT_1	POZ_CLIENT_2	POZ_PRODUS_1	POZ_PRODUS_2
Client 1 SRL	Produs 1	.324989	4	3	1	1
Client 1 SRL	Produs 2	411584	3	2	2	1
Client 1 SRL	Produs 5	4284714	2	1	3	2
Client 1 SRL	{SUBTOTAL client}	5021287	1	1	3	2
Client 2 SA	Produs 2	127530	1	1	4	3
Client 2 SA	{SUBTOTAL client}	127530	1	1	6	5
Client 3 SRL	Produs 2	41584	4	3	6	5
Client 3 SRL	Produs 3	33320	5	4	3	2
Client 3 SRL	Produs 4	84530	3	2	1	1
Client 3 SRL	Produs 5	5660235	2	1	2	1
Client 3 SRL	{SUBTOTAL client}	5819668	1	1	2	1
Client 5 SRL	Produs 2	95430	2	1	5	4
Client 5 SRL	Produs 3	58013	3	2	2	1
Client 5 SRL	{SUBTOTAL client}	153442	1	1	4	3
Client 7 SRL	Produs 2	136850	1	1	3	2
Client 7 SRL	{SUBTOTAL client}	136850	1	1	5	4
{SUBTOTAL produs}	Produs 1	324989	4	3	1	1
{SUBTOTAL produs}	Produs 2	812977	3	2	1	1
{SUBTOTAL produs}	Produs 3	91333	5	4	1	1
{SUBTOTAL produs}	Produs 4	84530	6	5	1	1
{SUBTOTAL produs}	Produs 5	9944949	2	1	1	1
{SUBTOTAL produs}	{SUBTOTAL client}	11258777	1	1	1	1

Figura 11.26. CUBE şi RANK, cu şi fără GROUPING

Care sunt cele mai mari cinci preţuri unitare din LINIIFACT?

În paragraful 9.2 am prezentat o soluţie ingenioasă bazată pe subconsultări în cascadă, iar în paragraful 9.4.2 variantă Oracle bazată pe ROWNUM. Din păcate, în alte servere de baze de date decât Oracle, pseudocoloana ROWNUM nu este operaţională. În schimb, DB2, Oracle şi SQL Server au implementat o nouă funcţie OLAP din SQL:1999, funcţie cu o logică similară - ROW_NUMBER -, astfel încât se poate redacta o soluţie comună al cărei rezultat este cel din stânga figurii 11.27.

```
SELECT PretUnit
FROM (SELECT PretUnit, ROW_NUMBER() OVER (ORDER BY PretUnit DESC)
                    AS Poz
          FROM liniifact) p1
WHERE Poz <=5
```

PRETUNIT
7064
7064
7064
7060
7060

PRETUNIT
7064
7060
6300
1705
1410

Figura 11.27. Două aplicări ale funcţiei ROW_NUMBER()

Funcţia ROW_NUMBER întoarce numărul de ordine al fiecărei linii în partiţia declarată, funcţionând deopotrivă în DB2, Oracle şi SQL Server. Dacă ne-ar interesa cele mai mari cinci preţuri unitare *distincte*, atunci am schimba interogarea de mai sus:

```
SELECT PretUnit
FROM
          (SELECT PretUnit, ROW_NUMBER() OVER (ORDER BY PretUnit DESC) AS
Poz
          FROM
                    (SELECT DISTINCT PretUnit
                    FROM liniifact
                    ) t1
          ) t2
WHERE Poz <=5
```

rezultatul fiind acum cel din dreapta figurii 11.27.

Care sunt cele mai mari cinci preţuri unitare de vânzare, produsele şi facturile în care apar cele cinci preţuri maxime?

Şi această soluţie este comună - DB2/Oracle/SQL Server:

```
SELECT DISTINCT NrFact, DenPr, lf.PretUnit
FROM liniifact lf
          INNER JOIN produse p ON lf.CodPr = p.CodPr
```

```
        INNER JOIN
                (SELECT Poz, PretUnit
                FROM
                        (SELECT PretUnit, ROW_NUMBER () OVER
                                (ORDER BY PretUnit DESC) AS Poz
                        FROM liniifact
                        ) p1
        WHERE Poz <=5) PMAX
                        ON lf.PretUnit >= PMAX.PretUnit
ORDER BY lf.PretUnit DESC
```

Să se afişeze ziua sau zilele în care s-au emis cele mai multe facturi.

O interogare comună, bazată pe funcţia analitică ROW_NUMBER, poate fi:

```
SELECT ROW_NUMBER() OVER (ORDER BY Nr_Facturilor DESC) AS Pozitie,
        DataFact AS Zi,
        Nr_Facturilor
FROM
        (SELECT DataFact, COUNT(*) AS Nr_Facturilor
        FROM facturi
        GROUP BY DataFact
        HAVING COUNT(*) >= ANY
                (SELECT Nr_Facturilor
                FROM
                (SELECT Nr_Facturilor, ROW_NUMBER() OVER
                (ORDER BY Nr_Facturilor DESC) AS Poz
                        FROM
                        (SELECT DISTINCT COUNT(*) AS Nr_Facturilor
                                FROM facturi
                                GROUP BY DataFact
                        ) X1
        ) X2
                WHERE X2.Poz <=1
                )
        ) X3
ORDER BY Nr_Facturilor DESC
```

Folosind funcţia RANK, ambele zile au acelaşi număr în top, 1. Şi noua variantă este valabilă atât în DB2 şi Oracle, cât şi în SQL Server.

```
SELECT *
FROM
        (SELECT RANK() OVER (ORDER BY COUNT(*) DESC) AS Pozitie,
```

```
                DataFact AS Zi,
                COUNT(*) AS Nr_Facturilor
          FROM facturi
          GROUP BY DataFact) X
WHERE Pozitie <= 1
```

Să se afişeze cele mai mari două valori ale numărului zilnic de facturi emise, precum şi datele în care s-au înregistrat respectivele valori.

Modificăm, în penultima interogare, ce utilizează funcţia ROW_NUMBER, doar 1 cu 2, adică, în loc de *X2.Poz <=1* vom scrie *X2.Poz <=2*. Însă, cu funcţia RANK, soluţia este (o idee) mai scurtă:

```
SELECT RANK() OVER (ORDER BY COUNT(*) DESC) AS Pozitie,
          DataFact AS Zi, COUNT(*) AS Nr_Facturilor
FROM facturi
GROUP BY DataFact
HAVING COUNT(*) >= ANY
          (SELECT Nr_Facturilor
          FROM
          (SELECT RANK() OVER (ORDER BY Nr_Facturilor DESC) AS Poz,
                    Nr_Facturilor
          FROM
                    (SELECT DISTINCT COUNT(*) AS Nr_Facturilor
                    FROM facturi
                    GROUP BY DataFact) X1
                    ) X2
          WHERE Poz <= 2
          )
```

POZITIE	ZI	NR_FACTURILOR
1	01-08-2007	4
1	07-08-2007	4
3	14-08-2007	3

Figura 11.28. Indicarea corectă a poziţiei din clasament

Care sunt cel mai bine vândute două produse ale fiecărei zile din luna august 2007?
Practic, ne interesează o listă precum cea din figura 11.29.

DATAFACT	DENPR	VINZARI	POZITIE_ZI
01-08-2007	Produs 5	10085012	1
01-08-2007	Produs 2	363570	2
02-08-2007	Produs 2	277950	1
03-08-2007	Produs 1	113050	1
03-08-2007	Produs 2	109000	2
04-08-2007	Produs 1	166005	1
04-08-2007	Produs 2	35970	2
07-08-2007	Produs 5	10246138	1
07-08-2007	Produs 2	253698	2
14-08-2007	Produs 5	4284714	1
14-08-2007	Produs 2	313375	2
15-08-2007	Produs 2	247757	1
16-08-2007	Produs 1	257159	1
16-08-2007	Produs 2	210261	2
21-08-2007	Produs 5	10283623	1
21-08-2007	Produs 2	159467	2

Figura 11.29. Cele mai bine vândute două produse din fiecare zi

Soluția următoare este una comună (cu excepția PostgreSQL și înlocuind, în DB2 și MS SQL Server, EXTRACT cu YEAR/MONTH):

```
SELECT *
FROM
        (SELECT DataFact, DenPr,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
                RANK() OVER (PARTITION BY DataFact
                        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
                AS Pozitie_Zi
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 8
        GROUP BY DataFact, DenPr) x1
WHERE Pozitie_Zi <= 2
ORDER BY DataFact, Pozitie_Zi
```

Asemănătoare funcției RANK este DENSE_RANK care, spre deosebire de prima, nu lasă goluri în clasament atunci când pe poziția precedentă sunt două sau mai multe linii. Diferența este pusă în valoare de interogarea următoare și rezultatul acesteia din figura 11.30.

```
SELECT DataFact, COUNT(*) AS Nr_Facturilor,
        RANK() OVER (ORDER BY COUNT(*) DESC) AS Poz_RANK,
        DENSE_RANK() OVER (ORDER BY COUNT(*) DESC)
                AS Poz_DENSE_RANK
```

```
FROM facturi
GROUP BY DataFact
```

DATAFACT	NR_FACTURILOR	POZ_RANK	POZ_DENSE_RANK
01-08-2007	4	1	1
07-08-2007	4	1	1
14-08-2007	3	3	2
16-08-2007	2	4	3
15-08-2007	2	4	3
01-09-2007	2	4	3
02-08-2007	2	4	3
21-08-2007	2	4	3
02-09-2007	2	4	3
10-09-2007	2	4	3
22-08-2007	1	11	4
03-08-2007	1	11	4
07-09-2007	1	11	4
17-09-2007	1	11	4
04-08-2007	1	11	4

Figura 11.30. Diferenţa dintre RANK şi DENSE_RANK

Cu DENSE_RANK se simplifică şi interogarea ce afişează cele mai mari două valori ale numărului zilnic de facturi emise în luna august 2007, precum şi datele (calendaristice) respective:

```
SELECT DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS Pozitie,
        DataFact AS Zi, COUNT(*) AS Nr_Facturilor
FROM facturi
WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
        EXTRACT (MONTH FROM DataFact) = 8
GROUP BY DataFact
HAVING COUNT(*) >= ANY
        (SELECT Nr_Facturilor
        FROM
                (SELECT DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS
        Poz,
                        COUNT(*) AS Nr_Facturilor
                FROM facturi
                WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                        EXTRACT (MONTH FROM DataFact) = 8
                GROUP BY DataFact
                ) X2
        WHERE Poz <= 2
        )
```

POZITIE	ZI	NR_FACTURILOR
1	01-08-2007	4
1	07-08-2007	4
2	14-08-2007	3

Figura 11.31. DENSE_RANK pentru obținerea primelor două valori ale numărului zilnic de facturi

În afara frecvent folositelor ROW_NUMBER, RANK și DENSE_RANK, în Oracle, funcțiile OLAP au fost gândite să-și apropie și mai mult o categorie profesională sedusă, de mult timp, de programe precum SPSS sau chiar Excel, și anume statisticienii. Apar astfel funcțiile: CUME_DIST, PERCENT_RANK[4], NTILE[5]. Pentru a le compara, folosim interogarea următoare al cărei rezultat este vizibil în figura 11.32.

POZITIE	ZI	NR_FACTURILOR	POZ_PROC_ZI	DISTRIB_CUME_ZI	TERTILE_ZI	CUARTILE_ZI
1	2007-08-01	4	0	0,13	1	1
1	2007-08-07	4	0	0,13	1	1
2	2007-08-14	3	0,14	0,2	1	1
3	2007-08-02	2	0,21	0,67	2	1
3	2007-09-01	2	0,21	0,67	2	2
3	2007-08-15	2	0,21	0,67	1	2
3	2007-08-16	2	0,21	0,67	1	2
3	2007-08-21	2	0,21	0,67	2	2
3	2007-09-10	2	0,21	0,67	2	3
3	2007-09-02	2	0,21	0,67	2	3
4	2007-08-22	1	0,71	1	3	3
4	2007-08-03	1	0,71	1	3	3
4	2007-09-17	1	0,71	1	3	4
4	2007-09-07	1	0,71	1	3	4
4	2007-08-04	1	0,71	1	3	4

Figura 11.32. Funcțiile Oracle PERCENT_RANK, CUME_DIST, și NTILE

```
SELECT
    DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS Pozitie,
    DataFact AS Zi,
    COUNT(*) AS Nr_Facturilor,
    ROUND(PERCENT_RANK() OVER (ORDER BY COUNT(*) DESC ),2)
        AS Poz_Proc_Zi,
    ROUND(CUME_DIST() OVER (ORDER BY COUNT(*) DESC),2)
        AS Distrib_Cume_Zi,
    NTILE(3) OVER (ORDER BY COUNT(*) DESC) AS Tertile_Zi,
    NTILE(4) OVER (ORDER BY COUNT(*) DESC) AS Cuartile_Zi
```

4. Funcțiile CUME_DIST și PERCENT_RANK nu sunt însă implementate nici în DB2, nici în MS SQL Server.
5. Funcția NTILE nu este implementată în DB2.

```
FROM facturi
    GROUP BY DataFact
```

- PERCENT_RANK întoarce poziţia procentuală a unei linii în cadrul unei partiţii, calculată prin formula: (poziţia liniei în partiţie - 1)/(numărul liniilor din partiţie - 1);
- CUME_DIST calculează poziţia unei valori specificate x relativă la o mulţime M de N valori: CUME_DIST(x) = numărul de valori (diferite sau egale cu x) din M ce îl precedă pe x (în ordinea specificată)/N;
- NTILE(3) calculează tertilele, iar NTILE(4) calculează cuartilele.

Din păcate, cunoştinţele mele de statistică nu s-au ameliorat de la prima ediţie a cărţii, aşa că ne oprim (tot) aici.

11.4. Ferestre pentru funcţiile analitice

Una dintre cele mai importante noţiuni introduse de Amendamentul OLAP al SQL:1999 este *fereastra*. Fereastra se defineşte în cadrul unei partiţii şi se referă la intervalul liniilor luat în calcul pentru linia curentă. Mărimea ferestrei se poate specifica fie fizic, printr-un număr de linii, fie logic, printr-un interval de tip dată calendaristică/timp sau interval de valori. Calculele se efectuează pentru fiecare linie din cadrul ferestrei, fereastră mişcătoare între o poziţie de start şi una de final. Linia curentă serveşte ca punct de referinţă pentru determinarea începutului şi sfârşitului ferestrei.

Specificaţiile unei ferestre privesc trei componente: partiţionarea, ordonarea şi grupurile de agregare. Orice funcţie de agregare poate fi utilizată în cadrul unei ferestre: SUM, AVG, MIN, MAX, STDDEV, VARIANCE, COUNT. Pe lângă acestea, Oracle (începând cu versiunea 8i) mai oferă suport pentru alte câteva funcţii statistice: VAR_SAMP, VAR_POP, STDDEV_SAMP ş.a.m.d., pe care, din lipsă de cunoştinţe de specialitate, nu le discutăm aici. Valorile agregate pot fi cumulative, mobile sau centrate.

Să se afişeze, pentru fiecare zi, valoarea cumulată a vânzărilor după fiecare factură emisă în luna septembrie 2007.

- Partiţia are dimensiunea unei zile; numărul de linii care o alcătuiesc depinde de numărul de facturi emise în ziua respectivă.
- Ordonarea în cadrul partiţiei se face după valoarea NrFact; prin urmare, poziţia de început a fiecărei partiţii este dată de prima factură din ziua respectivă, iar cea de sfârşit de factura ce are cel mai mare număr pentru DataFact ce defineşte partiţia curentă.
- Fereastra pentru care se efectuează calculul, pentru fiecare linie, are o margine fixă - începutul partiţiei şi una mobilă care este chiar linia curentă.

SQL:1999 a introdus clauza WINDOW în care se declară cele trei elemente enumerate anterior, partiționarea, ordonarea și grupurile de agregare. Soluția problemei formulate mai sus este:

```
SELECT DataFact AS Zi, NrFact, ValoareFact,
        SUM(ValoareFact) OVER W1 AS Val_Cumulata
FROM
        (SELECT DataFact, F.NrFact, SUM(Cantitate * PretUnit * (1 + ProcTVA))
                AS ValoareFact
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY DataFact, F.NrFact
        ) VALFACT
        WINDOW W1 AS (PARTITION BY DataFact ORDER BY NrFact
                                ROWS UNBOUNDED PRECEDING)
```

Fereastra definită are numele W1 și se constituie în cadrul partițiilor formate de valorile comune ale atributului DataFact. În cadrul partițiilor, liniile sunt ordonate crescător după valorile atributului NrFact, iar intervalul liniilor asupra căruia se aplică funcția de agregare SUM(...) este alcătuit din prima linie a partiției și linia curentă (*ROWS UNBOUNDED PRECEDING*).

Altă variantă SQL:1999 permite declararea „în linie" a ferestrei, variantă utilizată de DB2 și Oracle (înlocuind YEAR și MONTH cu EXTRACT-uri), parametrii ferestrei fiind definiți direct în clauza SELECT, după OVER (rezultatul este cel din figura 11.33):

```
SELECT DataFact AS Zi, NrFact, ValoareFact,
        SUM(ValoareFact) OVER (PARTITION BY DataFact
                ORDER BY NrFact ROWS UNBOUNDED PRECEDING)
                AS Val_Cumulata
FROM
        (SELECT DataFact, F.NrFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValoareFact
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
        GROUP BY DataFact, F.NrFact
        ) VALFACT
```

ZI	NRFACT	VALOAREFACT	VAL_CUMULATA
01.09.2007	3.111	4.442.960,0000	4.442.960,0000
01.09.2007	3.112	153.442,0000	4.596.402,0000
02.09.2007	3.113	127.530,0000	127.530,0000
02.09.2007	3.115	110.908,0000	238.438,0000
07.09.2007	3.119	5.819.668,0000	5.819.668,0000
10.09.2007	3.116	136.850,0000	136.850,0000
10.09.2007	3.117	287.855,0000	424.705,0000
17.09.2007	3.118	179.565,0000	179.565,0000

Figura 11.33. Valoarea cumulată, după fiecare factură, pentru fiecare zi

Din păcate, MS SQL Server este prea puțin generos în materie de ferestre. Există o clauză de partiționare, dar funcția-agregat (de obicei, SUM) nu se poate aplica decât la nivelul întregii partiții. Deci fereastra este chiar partiția. Ceea ce putem calcula este doar ceva de genul:

Care este contribuția fiecărei facturi emise în luna septembrie 2007 în totalul vânzărilor din ziua respectivă?

Fereastra este o zi (DataFact), rezultatul din figura 11.34 fiind furnizat de soluția comună SQL Server/DB2:

```
SELECT DataFact AS Zi, NrFact, ValoareFact,
        SUM(ValoareFact) OVER (PARTITION BY DataFact) AS Val_Cumulata
FROM
        (SELECT f.NrFact, DataFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValoareFact
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE YEAR(DataFact) = 2007 AND MONTH (DataFact) = 9
        GROUP BY f.NrFact, DataFact
        ) VALFACT
ORDER BY 1,2
```

Partea spumoasă - ferestrele „culisante" nu sunt la îndemână în MS SQL Server prin clauzele OVER/PARTITION.

Zi	NrFact	ValoareFact	Val_Cumulata
2007-09-01 00:00:00	3111	4442960.0000	4596402.0000
2007-09-01 00:00:00	3112	153442.0000	4596402.0000
2007-09-02 00:00:00	3113	127530.0000	238438.0000
2007-09-02 00:00:00	3115	110908.0000	238438.0000
2007-09-07 00:00:00	3119	5819668.0000	5819668.0000
2007-09-10 00:00:00	3116	136850.0000	424705.0000
2007-09-10 00:00:00	3117	287855.0000	424705.0000
2007-09-17 00:00:00	3118	179565.0000	179565.0000

Figura 11.34. O fereastră cât o zi

Să se calculeze (şi afişeze) soldul fiecărui client după fiecare vânzare şi încasare din luna august 2007.

Cele două operaţiuni care influenţează soldul (restul de plată al) unui client sunt vânzările (facturile emise) şi încasările. Orice vânzare creşte soldul, sold care indică cuantumul obligaţiilor de plată ale clientului către firma noastră, iar printr-o încasare un client îşi mai diminuează din obligaţii. Mai întâi, trebuie să obţinem o situaţie cronologică a facturărilor şi încasărilor, ceea ce presupune o reuniune (sintaxă Oracle):

```
SELECT F.NrFact, DataFact AS Data,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Facturat, 0 AS
Incasat
FROM produse p
        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 8
GROUP BY F.NrFact, DataFact
        UNION
SELECT NrFact, DataInc AS Data, 0 AS Facturat, Transa AS Incasat
FROM incasari i INNER JOIN incasfact if ON i.CodInc = if.CodInc
WHERE EXTRACT (YEAR FROM DataInc) = 2007 AND
                EXTRACT (MONTH FROM DataInc) = 8
```

Ultimele două coloane din rezultat (vezi figura 11.35) reprezintă cele două mărimi care influenţează soldul. Atributul Data reprezintă, după caz, fie data facturii, fie data încasării. În continuare, joncţionăm rezultatul acestei consultări cu tabela CLIENŢI, apoi construim o partiţie pentru fiecare client, iar soldul clientului va fi suma glisantă („mişcătoare") a expresiei *Facturat – Încasat*.

```
SELECT DenCl, Data, NrFact, Facturat, Incasat,
        SUM(Facturat-Incasat) OVER (PARTITION BY DenCl ORDER BY NrFact
                ROWS UNBOUNDED PRECEDING) AS Sold_Client
FROM (
        SELECT DenCl,t.*
        FROM
                (SELECT F.NrFact, DataFact AS Data,
                        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0)
                                AS Facturat, 0 AS Incasat
                FROM produse p
                        INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                        INNER JOIN facturi f ON lf.NrFact = f.NrFact
                WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                                EXTRACT (MONTH FROM DataFact) = 8
```

```
GROUP BY F.NrFact, DataFact
        UNION
SELECT NrFact, DataInc AS Data, 0 AS Facturat, Transa AS Incasat
FROM incasari i INNER JOIN incasfact if ON i.CodInc = if.CodInc
WHERE EXTRACT (YEAR FROM DataInc) = 2007 AND
                EXTRACT (MONTH FROM DataInc) = 8
) t INNER JOIN facturi f ON t.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
```

NRFACT	DATA	FACTURAT	INCASAT
1111	01-08-2007	4346038	0
1111	15-08-2007	0	53996
1112	01-08-2007	125516	0
1112	15-08-2007	0	125516
1113	01-08-2007	106275	0
1113	17-08-2007	0	106275
1114	01-08-2007	6021707	0
1115	02-08-2007	151238	0
1116	02-08-2007	126713	0
1117	03-08-2007	222050	0
1117	16-08-2007	0	9754
1117	17-08-2007	0	9754
1117	18-08-2007	0	3696
1118	04-08-2007	201975	0
1118	15-08-2007	0	101975
1118	16-08-2007	0	100000
1119	07-08-2007	5774499	0
1120	07-08-2007	[illegible]	0

Figura 11.35. Rezultat intermediar în calcularea soldului (pe client) după fiecare operaţiune

O bună parte din rezultat este prezentată în figura 11.36. Ultima coloană este suma culisantă a diferenţei dintre facturat şi încasat.

DENCL	NRFACT	DATA	FACTURAT	INCASAT	SOLD_CLIENT
Client 1 SRL	1111	01-08-2007	4346038	0	4346038
Client 1 SRL	1111	15-08-2007	0	53996	4292042
Client 1 SRL	1115	02-08-2007	151238	0	4443280
Client 1 SRL	1117	18-08-2007	0	3696	4439584
Client 1 SRL	1117	17-08-2007	0	9754	4429830
Client 1 SRL	1117	16-08-2007	0	9754	4420076
Client 1 SRL	1117	03-08-2007	222050	0	4642126
Client 1 SRL	1118	15-08-2007	0	101975	4540151
Client 1 SRL	1118	16-08-2007	0	100000	4440151
Client 1 SRL	1118	04-08-2007	201975	0	4642126
Client 1 SRL	1120	07-08-2007	97664	0	4739790
Client 1 SRL	1120	16-08-2007	0	7315	4732475
Client 1 SRL	2111	14-08-2007	4442960	0	9175435
Client 1 SRL	2115	15-08-2007	110908	0	9286343
Client 1 SRL	2117	16-08-2007	287855	0	9574198
Client 1 SRL	2118	16-08-2007	179565	0	9753763
Client 2 SA	1113	01-08-2007	106275	0	106275
Client 2 SA	1113	17-08-2007	0	106275	0
Client 2 SA	2113	14-08-2007	127530	0	127530
Client 3 SRL	1119	07-08-2007	5774499	0	5774499
Client 3 SRL	2119	21-08-2007	5819668	0	11594167
Client 4	1121	07-08-2007	4737838	0	4737838

Figura 11.36. Soldul clientului după fiecare operațiune (facturare sau încasare)

Interogarea următoare, funcționabilă în Oracle și DB2 (dacă înlocuim funcția EXTRACT) al cărei rezultat este cel din figura 11.37, ilustrează patru moduri de construire a unei ferestre:

- coloana *Fact_Precedte_Crta/Zi* calculează suma valorilor facturilor din ziua respectivă, începând cu prima și terminând cu factura curentă (*ROWS UNBOUNDED PRECEDING*);
- coloana *Fact_Crta_Precedta/Zi* însumează valoarea facturii curente și cea a precedentei (*ROWS 1 PRECEDING*) din ziua respectivă;
- la calculul *3Fact:_Ant_Crt_Urmat/Zi* se însumează valorile facturilor: curentă, precedentă și următoare (*ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING*);
- în fine, *Crta_Urmatoarele/Zi* însumează valoarea facturii curente și tuturor celor ce-i succedă în ziua repectivă (*ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING*).

```
SELECT DataFact AS Zi, NrFact, ValoareFact AS "Factura_Crt",
        SUM(ValoareFact) OVER  (PARTITION BY DataFact ORDER BY NrFact
                ROWS UNBOUNDED PRECEDING) AS "Fact_Precedte_Crta/Zi",
        SUM(ValoareFact) OVER (PARTITION BY DataFact ORDER BY NrFact
                ROWS 1 PRECEDING) AS "Fact_Crta_Precedta/Zi",
```

```
        SUM(ValoareFact) OVER (PARTITION BY DataFact ORDER BY NrFact
                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
                        AS "3Fact:_Ant_Crt_Urmat/Zi",
        SUM(ValoareFact) OVER (PARTITION BY DataFact ORDER BY NrFact
                ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
                        AS "Crta_Urmatoarele/Zi"
FROM
        (SELECT DataFact, F.NrFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS ValoareFact
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 8
        GROUP BY DataFact, F.NrFact
        ) VALFACT
```

ZI	NRFACT	Factura_Crt	Fact_Precedte_Crta/ZI	Fact_Crta_Precedta/ZI	3Fact:_Ant_Crt_Urmat/ZI	Crta_Urmatoarele/ZI
01-08-2007	1111	4346038	4346038	4346038	4471554	10599536
01-08-2007	1112	125516	4471554	4471554	4577829	6253498
01-08-2007	1113	106275	4577829	231791	6253498	6127982
01-08-2007	1114	6021707	10599536	6127982	6127982	6021707
02-08-2007	1115	151238	151238	151238	277951	277951
02-08-2007	1116	126713	277951	277951	277951	126713
03-08-2007	1117	222050	222050	222050	222050	222050
04-08-2007	1118	201975	201975	201975	201975	201975
07-08-2007	1119	5774499	5774499	5774499	5872163	10610001
07-08-2007	1120	97664	5872163	5872163	10610001	4835502
07-08-2007	1121	4737838	10610001	4835502	4835502	4737838
14-08-2007	2111	4442960	4442960	4442960	4596402	4723932
14-08-2007	2112	153442	4596402	4596402	4723932	280972
14-08-2007	2113	127530	4723932	280972	280972	127530
15-08-2007	2115	110908	110908	110908	247758	247758
15-08-2007	2116	136850	247758	247758	247758	136850
16-08-2007	2117	287855	287855	287855	467420	467420
16-08-2007	2118	179565	467420	467420	467420	179565
21-08-2007	2119	5819668	5819668	5819668	10560940	10560940
21-08-2007	2121	4741272	10560940	10560940	10560940	4741272

Figura 11.37. Patru moduri de definire a ferestrelor

Dacă în SELECT-ul precedent mărimea fereastrelor era definită fizic, prin numărul de linii ce precedă sau succedă linia curentă, în următoarea interogare se pun faţă în faţă ferestre definite fizic, prin număr de linii, şi logic, prin intervale (de zile).

```
SELECT Client, DataFact AS Zi, Vanzari,
        SUM(Vanzari) OVER  (PARTITION BY Client ORDER BY DataFact
                ROWS 1 PRECEDING) AS Crt_Prec1_Linii,
```

```
        SUM(Vanzari) OVER (PARTITION BY Client ORDER BY DataFact
                RANGE INTERVAL '1' DAY PRECEDING) AS Crt_Prec1_Zile,
        SUM(Vanzari) OVER (PARTITION BY Client ORDER BY DataFact
                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
                        AS P1_Act_U1_Linii,
        SUM(Vanzari) OVER (PARTITION BY Client ORDER BY DataFact
                RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
                        INTERVAL '1' DAY FOLLOWING) AS P1_Act_U1_Zile
FROM
        (SELECT DenCl AS Client, DataFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
                INNER JOIN clienti c ON f.CodCl = c.CodCl
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY DenCl, DataFact
        ) VALFACT
```

- coloana *Crt_Prec1_Linii* însumează, după cum am văzut în precedenta interogare, valoarea facturii curente şi pe cea a precedentei (*ROWS 1 PRECEDING*);
- la calcului valorii *Crt_Prec1_Zile* se ia în calcul linia precedentă, numai dacă data căruia îi este asociată este imediat înaintea datei din linia curentă (*RANGE INTERVAL '1' DAY PRECEDING*) - vezi, pentru Client 1, situaţia zilei de 7 august;

CLIENT	ZI	VINZARI	CRT_PREC1_LINII	CRT_PREC1_ZILE	P1_ACT_U1_LINII	P1_ACT_U1_ZILE
Client 1 SRL	01-09-2007	4442960	4442960	4442960	4553868	4553868
Client 1 SRL	02-09-2007	110908	4553868	4553868	4841723	4553868
Client 1 SRL	10-09-2007	287855	398763	287855	578326	287855
Client 1 SRL	17-09-2007	179565	467420	179565	467420	179565
Client 2 SA	02-09-2007	127530	127530	127530	127530	127530
Client 3 SRL	07-09-2007	5819668	5819668	5819668	5819668	5819668
Client 5 SRL	01-09-2007	153442	153442	153442	153442	153442
Client 7 SRL	10-09-2007	136850	136850	136850	136850	136850

Figura 11.38. Ferestelor definite logic şi fizic

Curios, deşi interogarea respectă specificaţiile SQL-99 şi funcţionează în Oracle, în DB2 nu am reuşit să o fac operaţională.

11.5. Comparaţii şi ponderi

Atunci când, într-o partiţie, se doreşte raportarea valorilor liniei curente la cele din prima sau ultima linie dintr-un interval sau fereastră, este rezonabil să se folosească funcţiile FIRST_VALUE şi LAST_VALUE implementate în DB2 şi Oracle, nu însă şi în SQL Server.

Care este raportul zilnic al vânzărilor pe luna septembrie 2007, relativ la prima zi de vânzări?

```
SELECT
        DataFact AS Zi,Vanzari,
        FIRST_VALUE(Vanzari) OVER (ORDER BY DataFact) AS Vnz_Zi1,
        ROUND(Vanzari/FIRST_VALUE(Vanzari) OVER (ORDER BY DataFact),2)
                AS Raport_Crt_Prima_Zi
FROM
        (SELECT DataFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY DataFact
        )
```

ZI	VINZARI	VNZ_ZI1	RAPORT_CRT_PRIMA_ZI
01-09-2007	4596402	4596402	1
02-09-2007	238438	4596402	0,05
07-09-2007	5819668	4596402	1,27
10-09-2007	424705	4596402	0,09
17-09-2007	179565	4596402	0,04

Figura 11.39. Exemplu de utilizare a funcţiei FIRST_VALUE

Să se determine zilele din luna august 2007 în care s-a vândut cel mai bine fiecare produs.

Una dintre soluţii se bazează pe o subconsultare ce întrebuinţează funcţia FIRST_VALUE:

```
SELECT DenPr, Datafact, Vanzari
FROM
        (SELECT DenPr, DataFact,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
                FIRST_VALUE(ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0))
                        OVER (PARTITION BY DenPr ORDER BY
                        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) DESC)
```

```
            AS Vnz_Max
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
            INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
        EXTRACT (MONTH FROM DataFact) = 8
        GROUP BY DenPr, DataFact
        ) t
WHERE Vanzari = Vnz_Max
```

DENPR	DATAFACT	VINZARI	VNZ_MAX
Produs 1	16-08-2007	257159	257159
Produs 1	04-08-2007	166005	257159
Produs 1	03-08-2007	113050	257159
Produs 1	14-08-2007	67830	257159
Produs 1	01-08-2007	59500	257159
Produs 2	01-08-2007	363570	363570
Produs 2	14-08-2007	313375	363570
Produs 2	02-08-2007	277950	363570
Produs 2	07-08-2007	253698	363570
Produs 2	15-08-2007	247757	363570
Produs 2	16-08-2007	210261	363570
Produs 2	21-08-2007	159467	363570
Produs 2	03-08-2007	109000	363570
Produs 2	04-08-2007	35970	363570
Produs 3	14-08-2007	58013	58013
Produs 3	01-08-2007	35700	58013
Produs 3	21-08-2007	33320	58013
Produs 3	07-08-2007	33320	58013
Produs 4	21-08-2007	84530	84530
Produs 4	07-08-2007	76845	84530
Produs 4	01-08-2007	55754	84530
Produs 5	21-08-2007	10283623	10283623
Produs 5	07-08-2007	10246138	10283623
Produs 5	01-08-2007	10085012	10283623
Produs 5	14-08-2007	4284714	10283623

Figura 11.40. Determinarea, pentru fiecare produs, a vânzărilor şi a maximului zilnic din luna august 2007

Subconsultarea:

```
SELECT DenPr, DataFact,
        ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari,
        FIRST_VALUE(ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0))
            OVER (PARTITION BY DenPr ORDER BY
```

```
            ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) DESC)
        AS Vnz_Max
FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
EXTRACT (MONTH FROM DataFact) = 8
GROUP BY DenPr, DataFact
```

obţine rezultatul din figura 11.40. În final, răspunsul problemei este cel din figura 11.41.

DENPR	DATAFACT	VINZARI
Produs 1	16-08-2007	257159
Produs 2	01-08-2007	363570
Produs 3	14-08-2007	58013
Produs 4	21-08-2007	84530
Produs 5	21-08-2007	10283623

Figura 11.41. Zilele în care s-a vândut cel mai bine fiecare produs

Pentru respectarea adevărului istoric, se cuvine de remarcat că, în locul cosmopolitei funcţii FIRST_VALUE, am fi putut folosi în subconsultare şi tocita MAX, bineînţeles, într-o haină nouă:

```
SELECT DenPr, DataFact, ROUND(Vanzari,0) AS Vanzari
FROM
        (SELECT DenPr, DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari,
                MAX(SUM(Cantitate * PretUnit * (1 + ProcTVA))) OVER
                        (PARTITION BY DenPr
                        ORDER BY SUM(Cantitate * PretUnit * (1 + ProcTVA)) DESC)
                AS Vnz_Max
        FROM produse p INNER JOIN liniifact lf ON p.CodPr = lf.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 8
        GROUP BY DenPr, DataFact
        )
WHERE Vanzari = Vnz_Max
```

Atunci când se doreşte determinarea: contribuţiei unui produs la totalul vânzărilor, părţii unui client în totalul datornicilor, procentului unui produs din valoarea unei facturi, ponderii salariului directorului general (sau al şefilor de compartimente) în totalul fondului de salarii al unei întreprinderi ş.a.m.d. se poate apela la serviciile funcţiei RATIO_TO_REPORT care calculează raportul dintre o valoare (atribut/expresie) şi suma totală a valorii respective pentru toate liniile din fereastră. Funcţia este implementată în Oracle, nu şi în DB2 şi MS SQL Server.

Să se afişeze structura vânzărilor pe judeţe.

```
SELECT Judet, Vanzari,
        ROUND(RATIO_TO_REPORT(Vanzari) OVER (),2) AS Pondere_Judet
FROM
        (SELECT Judet,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM judete j
                        INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                        INNER JOIN clienti c ON cp.CodPost = c.CodPost
                        INNER JOIN facturi f ON c.CodCl = f.CodCl
                        INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY Judet
        )
```

După cum se observă, avantajul funcţiei este că nu necesită precizarea explicită a numitorului. Rezultatul interogării este prezentat în figura 11.42.

JUDET	VINZARI	PONDERE_JUDET
Iasi	24901983	0,51
Neamt	6021707	0,12
Timis	832812	0,02
Vaslui	17413835	0,35

Figura 11.42. Contribuţia fiecărui judeţ în totalul vânzărilor

Să se calculeze contribuţia produselor la valoarea fiecărei facturi emise în luna septembrie 2007.

```
SELECT Factura, Produs, Vanzari,
        ' '|| ROUND(RATIO_TO_REPORT(Vanzari) OVER (PARTITION BY Factura),2)
                * 100 ||'%' AS Pondere_Prod_Fact
FROM
        (SELECT NrFact AS Factura, DenPr AS Produs,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM produse p INNER JOIN liniifact lf ON P.CodPr = LF.Codpr
                        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY NrFact, DenPr
        )
```

FACTURA	PRODUS	VINZARI	PONDERE_PROD_FACT
3111	Produs 1	67830	2%
3111	Produs 2	90416	2%
3111	Produs 5	4284714	96%
3112	Produs 2	95430	62%
3112	Produs 3	58013	38%
3113	Produs 2	127530	100%
3115	Produs 2	110908	100%
3116	Produs 2	136850	100%
3117	Produs 1	124355	43%
3117	Produs 2	163500	57%
3118	Produs 1	132804	74%
3118	Produs 2	46761	26%
3119	Produs 2	41584	1%
3119	Produs 3	33320	1%
3119	Produs 4	84530	1%
3119	Produs 5	5660235	97%

Figura 11.43. Ponderea fiecărui produs în fiecare factură emisă în luna septembrie 2007

La analiza dinamicii unei mărimi, se raportează valoarea din linia curentă la cea dintr-o altă linie plasată la o anumită distanţă. Spre exemplu, pentru compararea vânzărilor fiecărei luni calendaristice la aceeaşi lună din anul precedent, va trebui să împărţim valoarea de pe linia curentă la o valoare aflată cu 12 linii/luni înainte. Apare astfel noţiunea de *deplasament înapoi* (LAG) şi *înainte* (LEAD). Fireşte, soluţia funcţionează numai dacă nu există luni de întrerupere, adică intervalul este continuu. În interogarea următoare, al cărei rezultat este prezentat în figura 11.44, deplasamentul este, pentru ambele funcţii, 1.

```
SELECT Zi, Vanzari,
        LAG(Vanzari,1) OVER (ORDER BY Zi) AS Vnz_LAG,
        LEAD(Vanzari,1) OVER (ORDER BY Zi) AS Vnz_LEAD
FROM
        (SELECT DataFact AS Zi,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM produse p
                INNER JOIN liniifact lf ON P.CodPr = LF.Codpr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY DataFact)
```

Funcţiile LAG şi LEAD extrag vânzările din ziua precedentă, respectiv următoare. Locurile libere din cadrul listei reprezintă valori NULL. Ambele funcţii sunt implementate în DB2 şi Oracle, nu însă şi în MS SQL Server.

ZI	VINZARI	VNZ_LAG	VNZ_LEAD
01-09-2007	4596402	(null)	238438
02-09-2007	238438	4596402	5819668
07-09-2007	5819668	238438	424705
10-09-2007	424705	5819668	179565
17-09-2007	179565	424705	(null)

Figura 11.44. Vânzările din ziua curentă, precedentă şi următoare

Care este evoluția vânzărilor de la o zi la alta, pentru luna septembrie 2007?

```
SELECT Zi, Vanzari,
        COALESCE(LAG(Vanzari,1) OVER (ORDER BY Zi),0) AS Vnz_Preced,
        Vanzari - COALESCE((LAG(Vanzari,1) OVER (ORDER BY Zi)),0) AS Diferenta
FROM
        (SELECT DataFact AS Zi,
                ROUND(SUM(Cantitate * PretUnit * (1 + ProcTVA)),0) AS Vanzari
        FROM produse p
                INNER JOIN liniifact lf ON P.CodPr = LF.Codpr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT (MONTH FROM DataFact) = 9
        GROUP BY DataFact
        )
```

ZI	VINZARI	VNZ_PRECED	DIFERENTA
01-09-2007	4596402	0	4596402
02-09-2007	238438	4596402	-4357964
07-09-2007	5819668	238438	5581230
10-09-2007	424705	5819668	-5394963
17-09-2007	179565	424705	-245140

Figura 11.45. Evoluția zilnică a vânzărilor pentru septembrie 2007

Capitolul 12
SQL, ierarhii şi recursivitate

Problemele grafurilor, ierarhiilor şi recursivităţii în SQL reprezintă un subiect generos despre care au scris, printre alţii, Joe Celko ([Celko 2004], [Celko 2005]), Stephane Faroult [Faroult & Robson 2006], Anthony Molinaro ([Molinaro 2006]), Fabian Pascal ([Pascal 2000]) şi Vadim Tropashko ([Tropashko 2005]). Faroult şi Robson inventariază trei modele de reprezentare a ierarhiilor:

- modelul adiacent - similar modului de reprezentare a şefilor direcţi în tabela PERSONAL2 (figura 8.4);
- modelul „căii materializate", în care simbolul unui nod indică poziţia sa în ierarhie, cum ar fi cazul planului de conturi din contabilitatea unei companii (de exemplu, 401 - Furnizori, 401.001 - Furnizorul A, 401.002 - Furnizorul B etc.)[1];
- modelul setului imbricat (vezi [Celko 2004], [Celko 2005]), în care fiecărui nod îi sunt asociate o pereche de numere care definesc intervalul în care sunt plasaţi subordonaţii săi.

În acest capitol ne vom ocupa fugitiv doar de primul model şi câteva alte probleme circumscrise grafurilor şi ierarhiilor.

12.1. Autojoncţiuni şi subconsultări

Pentru aflarea majorităţii informaţiilor privitoare la ierarhia firmei, soluţia obişnuită în SQL constă în joncţionarea a două instanţe ale tabelei PERSONAL2 (P1 şi P2) după condiţia *P1.MarcaSef = P2.Marca*.

Prin interogarea:

```
SELECT *
FROM personal2 p1 INNER JOIN personal2 p2 ON p1.MarcaSef = p2.Marca
```

se obţine un rezultat de forma celui din figura 12.1. Primele şase coloane corespund primei instanţe, P1, în timp ce restul, celei de-a doua instanţe, P2. P1 este legată de calitatea de subordonat, iar P2 de cea de şef. De aceea, ar fi mai nimerită redenumirea lui P1 ca *SUBORDONAŢI*, iar a lui P2 drept *ŞEFI*.

1. Pentru detalii, vezi şi [Fotache 2005], p. 196.

marca	numepren	datanast	compart	marcasef	saltarifar	marca	numepren	datanast	compart	marcasef	saltarifar
2	ANGAJAT 2	1977-10-11 00:00:00	FINANCIAR	1	1450.00	1	ANGAJAT 1	1962-07-01 00:00:00	DIRECTIUNE	NULL	1600.00
3	ANGAJAT 3	1982-08-02 00:00:00	MARKETING	1	1450.00	1	ANGAJAT 1	1962-07-01 00:00:00	DIRECTIUNE	NULL	1600.00
4	ANGAJAT 4	NULL	FINANCIAR	2	1380.00	2	ANGAJAT 2	1977-10-11 00:00:00	FINANCIAR	1	1450.00
5	ANGAJAT 5	1985-04-30 00:00:00	FINANCIAR	2	1420.00	2	ANGAJAT 2	1977-10-11 00:00:00	FINANCIAR	1	1450.00
6	ANGAJAT 6	1985-11-09 00:00:00	FINANCIAR	5	1350.00	5	ANGAJAT 5	1965-04-30 00:00:00	FINANCIAR	2	1420.00
7	ANGAJAT 7	NULL	FINANCIAR	5	1280.00	5	ANGAJAT 5	1965-04-30 00:00:00	FINANCIAR	2	1420.00
8	ANGAJAT 8	1980-12-31 00:00:00	MARKETING	3	1290.00	3	ANGAJAT 3	1962-08-02 00:00:00	MARKETING	1	1450.00
9	ANGAJAT 9	1976-02-28 00:00:00	MARKETING	3	1410.00	3	ANGAJAT 3	1962-08-02 00:00:00	MARKETING	1	1450.00
10	ANGAJAT 10	1972-01-29 00:00:00	RESURSE UMANE	1	1370.00	1	ANGAJAT 1	1962-07-01 00:00:00	DIRECTIUNE	NULL	1600.00

Figura 12.1. Autojoncţiunea tabelei PERSONAL2

Să luăm în discuţie câteva probleme.

Cum se numeşte şeful Angajatului 7?

Acum chiar denumim cele două instanţe ale tabelei PERSONAL2 SUBORDONAŢI, respectiv ŞEFI, pentru a limpezi atât specificarea condiţiei de joncţiune, cât şi formularea predicatului de selecţie şi a coloanei de afişat (rezultatul este în figura 12.2).

```
SELECT sefi.NumePren
FROM personal2 subordonati
INNER JOIN personal2 sefi ON subordonati.MarcaSef = sefi.Marca
WHERE subordonati.NumePren = 'ANGAJAT 7'
```

NumePren
ANGAJAT 5

Figura 12.2. Şeful direct al Angajatului 7

Care sunt subordonaţii direcţi ai Angajatului 2?

Păstrăm, din interogarea anterioară, titulaturile celor două instanţe, modificând predicatul şi numele coloanei extrase în rezultat (rezultatul este în figura 12.3):

```
SELECT subordonati.NumePren
FROM personal2 subordonati
INNER JOIN personal2 sefi ON subordonati.MarcaSef = sefi.Marca
WHERE sefi.NumePren = 'ANGAJAT 2'
```

NumePren
ANGAJAT 4
ANGAJAT 5

Figura 12.3. Subordonaţii direcţi ai Angajatului 2

Fireşte, cele două probleme pot fi rezolvate şi prin subconsultări, astfel:

```
SELECT NumePren
FROM personal2
WHERE Marca IN
        (SELECT MarcaSef
        FROM personal2
```

```
        WHERE NumePren = 'ANGAJAT 7')
```

respectiv:

```
SELECT NumePren
FROM personal2
WHERE MarcaSef IN
        (SELECT Marca
        FROM personal2
        WHERE NumePren = 'ANGAJAT 2')
```

Câţi subordonaţi are fiecare angajat al firmei?

Soluţia:

```
SELECT sefi.NumePren, COUNT(*) AS Nr_Subordonati
FROM personal2 subordonati
INNER JOIN personal2 sefi ON subordonati.MarcaSef = sefi.Marca
GROUP BY sefi.NumePren
```

îi extrage numai pe cei care au măcar un subordonat. Dacă dorim ca rezultatul să conţină toţi angajaţii în ordine alfabetică (figura 12.4), trebuie folosită joncţiunea externă la dreapta şi modificat argumentul funcţiei COUNT:

```
SELECT sefi.NumePren,
COUNT(SUBORDONATI.MarcaSef) AS Nr_Subordonati
FROM personal2 subordonati RIGHT OUTER JOIN personal2 sefi
ON subordonati.MarcaSef = sefi.Marca
GROUP BY sefi.NumePren
ORDER BY 1
```

Figura 12.4 conţine urmările acestei interogări.

NumePren	Nr_Subordonati
ANGAJAT 1	3
ANGAJAT 10	0
ANGAJAT 2	2
ANGAJAT 3	2
ANGAJAT 4	0
ANGAJAT 5	2
ANGAJAT 6	0
ANGAJAT 7	0
ANGAJAT 8	0
ANGAJAT 9	0

Figura 12.4. Numărul subordonaţilor pentru fiecare salariat

Care sunt subordonaţii subordonaţilor directorului general?

Din punctul de vedere al arborelui ce oglindeşte ierarhia firmei (figura 12.5), ne interesează „nepoţii rădăcinii". Interogarea trebuie să parcurgă trei nivele ierarhice, şi, în final, să extragă salariaţii cu mărcile 4, 5, 8 şi 9.

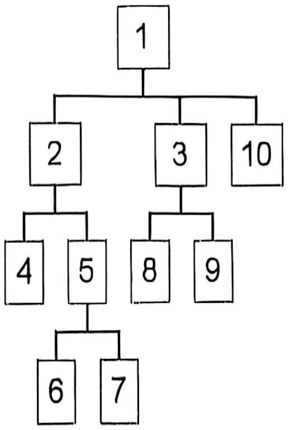


Figura 12.5. Ierarhia firmei

Varianta 1 (joncţiuni):

```
SELECT subordonati.Marca, subordonati.NumePren, subordonati.Compart
FROM personal2 subordonati
        INNER JOIN personal2 sefi1 ON subordonati.MarcaSef = sefi1.Marca
        INNER JOIN personal2 sefi2 ON sefi1.MarcaSef = sefi2.Marca
WHERE sefi2.MarcaSef IS NULL
```

Varianta 2 (subconsultări):

```
SELECT Marca, NumePren, Compart
FROM personal2
WHERE MarcaSef IN
        (SELECT Marca
        FROM personal2
        WHERE MarcaSef IN
                (SELECT Marca
                FROM personal2
                WHERE MarcaSef IS NULL)
        )
```

Rezultatul (figura 12.6) confirmă bănuielile noastre.

Marca	NumePren	Compart
4	ANGAJAT 4	FINANCIAR
5	ANGAJAT 5	FINANCIAR
8	ANGAJAT 8	MARKETING
9	ANGAJAT 9	MARKETING

Figura 12.6. Subordonaţii direcţi ai subordonaţilor „imediaţi" directorului general

Care este nivelul ierarhic al fiecărui salariat?

De la început, ştim că ierarhia reflectată în tabela PERSONAL2 se derulează pe patru nivele, aşa încât o interogare pentru rezolvarea problemei poate fi:

```
SELECT NumePren, Compart, 'Nivel 1' AS Nivel
FROM personal2
WHERE MarcaSef IS NULL
          UNION
SELECT NIVEL2.NumePren, NIVEL2.Compart, 'Nivel 2' AS Nivel
FROM personal2 NIVEL1 INNER JOIN personal2 NIVEL2
          ON NIVEL1.Marca = NIVEL2.MarcaSef AND NIVEL1.MarcaSef IS NULL
          UNION
SELECT NIVEL3.NumePren, NIVEL3.Compart, 'Nivel 3' AS Nivel
FROM personal2 NIVEL1 INNER JOIN personal2 NIVEL2
          ON NIVEL1.Marca = NIVEL2.MarcaSef
          AND NIVEL1.MarcaSef IS NULL
          INNER JOIN personal2 NIVEL3 ON NIVEL2.Marca = NIVEL3.MarcaSef
          UNION
SELECT NIVEL4.NumePren, NIVEL4.Compart, 'Nivel 4' AS Nivel
FROM personal2 NIVEL1
          INNER JOIN personal2 NIVEL2 ON NIVEL1.Marca = NIVEL2.MarcaSef
                    AND NIVEL1.MarcaSef IS NULL
          INNER JOIN personal2 NIVEL3 ON NIVEL2.Marca = NIVEL3.MarcaSef
          INNER JOIN personal2 NIVEL4 ON NIVEL3.Marca = NIVEL4.MarcaSef
ORDER BY NumePren
```

Iată şi rezultatul în figura 12.7. Interogarea funcţionează rezonabil. Există, însă, cel puţin două umbre: trebuie să ştim, aprioric, numărul nivelelor ierarhice, iar în al doilea rând, la un număr mult mai mare de nivele, întinderea consultării creşte sensibil.

NumePren	Compart	Nivel
ANGAJAT 1	DIRECTIUNE	Nivel 1
ANGAJAT 10	RESURSE UMANE	Nivel 2
ANGAJAT 2	FINANCIAR	Nivel 2
ANGAJAT 3	MARKETING	Nivel 2
ANGAJAT 4	FINANCIAR	Nivel 3
ANGAJAT 5	FINANCIAR	Nivel 3
ANGAJAT 6	FINANCIAR	Nivel 4
ANGAJAT 7	FINANCIAR	Nivel 4
ANGAJAT 8	MARKETING	Nivel 3
ANGAJAT 9	MARKETING	Nivel 3

Figura 12.7. Nivelul ierarhic al fiecărui angajat

O variantă mai elegantă, ca efort de scriere este:

```
SELECT NIVEL4.NumePren, NIVEL4.Compart,
       CASE
```

```
        WHEN NIVEL4.MarcaSef IS NULL THEN 1
              WHEN NIVEL3.MarcaSef IS NULL THEN 2
              WHEN NIVEL2.MarcaSef IS NULL THEN 3
              WHEN NIVEL1.MarcaSef IS NULL THEN 4
        END AS Nivel
FROM personal2 NIVEL1
        RIGHT OUTER JOIN personal2 NIVEL2
              ON NIVEL1.Marca = NIVEL2.MarcaSef AND NIVEL1.MarcaSef IS
        NULL
        RIGHT OUTER JOIN personal2 NIVEL3
              ON NIVEL2.Marca = NIVEL3.MarcaSef
        RIGHT OUTER JOIN personal2 NIVEL4
              ON NIVEL3.Marca = NIVEL4.MarcaSef
```

Se joncţionează extern patru instanţe ale tabelei PERSONAL2, câte una pentru fiecare nivel ierarhic, astfel încât au şi fost denumite NIVEL1, ..., NIVEL4. Fraza SELECT compune ierarhia pornind de la NIVEL1 care reprezintă o instanţă a PERSONAL2 cu o singură linie, cea a directorului general, instanţă joncţionată extern la dreapta cu NIVEL2 care va furniza subordonaţii direcţi ai înregistrării-rădăcină ş.a.m.d.

Să se afişeze structura ierarhică a firmei.

Practic, dorim o formă de vizualizare a angajaţilor care să ţină cont de modul lor de subordonare – ca în figura 12.8.

NUME	COMPART	SEF1	SEF2	SEF3	SEF4
ANGAJAT 1	DIRECTIUNE	1	1	1	1
--ANGAJAT 2	FINANCIAR	1	2	2	2
-- --ANGAJAT 4	FINANCIAR	1	2	4	4
-- --ANGAJAT 5	FINANCIAR	1	2	5	5
-- -- --ANGAJAT 6	FINANCIAR	1	2	5	6
-- -- --ANGAJAT 7	FINANCIAR	1	2	5	7
--ANGAJAT 3	MARKETING	1	3	3	3
-- --ANGAJAT 8	MARKETING	1	3	8	8
-- --ANGAJAT 9	MARKETING	1	3	9	9
--ANGAJAT 10	RESURSE UMANE	1	10	10	10

Figura 12.8. Vizualizarea ierarhiei

Dacă până acum, interogările erau executabile în toate serverele luate în discuţie în această carte, următoarea funcţionează în Oracle, DB2 şi, cu uşoare modificări (operatorul de concatenare | | se înlocuieşte cu +), şi în MS SQL Server.

```
WITH nivele AS
        (SELECT NIVEL4.*,
              CASE
```

```
                    WHEN NIVEL4.MarcaSef IS NULL THEN 1
                    WHEN NIVEL3.MarcaSef IS NULL THEN 2
                    WHEN NIVEL2.MarcaSef IS NULL THEN 3
                    WHEN NIVEL1.MarcaSef IS NULL THEN 4
                END AS Nivel
        FROM personal2 NIVEL1
                RIGHT OUTER JOIN personal2 NIVEL2
                    ON NIVEL1.Marca = NIVEL2.MarcaSef
                        AND Nivel1.MarcaSef IS NULL
                RIGHT OUTER JOIN personal2 NIVEL3
                    ON NIVEL2.Marca = NIVEL3.MarcaSef
                RIGHT OUTER JOIN personal2 NIVEL4
                    ON NIVEL3.Marca = NIVEL4.MarcaSef )
SELECT NumePren AS Nume, Compart, Marca AS Sef1,
        Marca AS Sef2, Marca AS Sef3, Marca AS Sef4
FROM nivele
WHERE Nivel = 1
        UNION
SELECT '–'||N2.NumePren AS Nume, N2.Compart, N1.Marca AS Sef1,
        N2.Marca AS Sef2, N2.Marca AS Sef3, N2.Marca AS Sef4
FROM nivele N2 INNER JOIN nivele N1
                ON N2.Nivel = 2 AND N2.MarcaSef = N1.Marca AND N1.Nivel = 1
        UNION
SELECT '––'||N3.NumePren AS Nume, N3.Compart, N1.Marca AS Sef1,
        N2.Marca AS Sef2, N3.Marca AS Sef3, N3.Marca AS Sef4
FROM nivele N3 INNER JOIN nivele N2
                ON N3.Nivel = 3 AND N3.MarcaSef = N2.Marca AND N2.Nivel = 2
        INNER JOIN nivele N1
                ON N2.Nivel = 2 AND N2.MarcaSef = N1.Marca AND N1.Nivel = 1
        UNION
SELECT '–––'||N4.NumePren AS Nume, N4.Compart, N1.Marca AS Sef1,
        N2.Marca AS Sef2, N3.Marca AS Sef3, N4.Marca AS Sef4
FROM nivele N4
        INNER JOIN nivele N3
                ON N4.Nivel = 4 AND N4.MarcaSef = N3.Marca AND N3.Nivel = 3
        INNER JOIN nivele N2
                ON N3.Nivel = 3 AND N3.MarcaSef = N2.Marca AND N2.Nivel = 2
        INNER JOIN nivele N1
                ON N2.Nivel = 2 AND N2.MarcaSef = N1.Marca AND N1.Nivel = 1
ORDER BY Sef1, Sef2, Sef3, Sef4
```

Lucrurile pot fi simplificate sensibil utilizând o structură de tip CASE prin care determinăm nivelul de subordonare, nivel care va determina de câte ori se repetă (REPEAT în DB2, LPAD în Oracle şi PostgreSQL, REPLICATE în MS SQL Server) grupul de liniuțe, astfel încât listarea primei coloane este asemănătoare celei din figura 12.8. Iată varianta SQL Server:

```
SELECT REPLICATE ('– ', 1 * ( (
        CASE
                WHEN NIVEL4.MarcaSef IS NULL THEN 1
                WHEN NIVEL3.MarcaSef IS NULL THEN 2
                WHEN NIVEL2.MarcaSef IS NULL THEN 3
                WHEN NIVEL1.MarcaSef IS NULL THEN 4
        END ) - 1 ) ) + NIVEL4.NumePren AS Nume, NIVEL4.Compart
FROM personal2 NIVEL1
        RIGHT OUTER JOIN personal2 NIVEL2
                ON NIVEL1.Marca = NIVEL2.MarcaSef AND
                        NIVEL1.MarcaSef IS NULL
        RIGHT OUTER JOIN personal2 NIVEL3
                        ON NIVEL2.Marca = NIVEL3.MarcaSef
        RIGHT OUTER JOIN personal2 NIVEL4
                        ON NIVEL3.Marca = NIVEL4.MarcaSef
```

O problemă grozav de interesantă pentru ceea ce înseamnă ierarhie şi recursivitate în SQL este cea tratată în paragraful 8.3 al cărții dedicate proiectării (şi implementării) bazelor de date ([Fotache 2005]). Pornind de la o tabelă DISTANȚE {Loc1, Loc2, Distanța} care conține numărul de kilometri dintre două localități oarecum vecine între care există o şosea „practicabilă" (vorba vine!), putem să determinăm traseele posibile dintre două oraşe. Lucrurile nu sunt atât de simple, însă, fiind necesare joncțiuni repetate corespunzătoare „punctelor intermediare". Pentru exemplificare (sintaxă Oracle), intimidanta interogare de mai jos extrage toate traseele posibile dintre Iaşi şi Focşani, prin maximum cinci localități intermediare:

```
WITH distante_tot AS
        (SELECT Loc1, Loc2, Distanta FROM distante
                UNION
        SELECT Loc2, Loc1, Distanta FROM distante)
– prima este (eventuala) rută directă
SELECT d1.Loc1 AS Plecare, d1.Loc2 AS Sosire,
        d1.Loc1 || '**' || d1.Loc2 AS Sir,
        Distanta
FROM distante_tot d1
WHERE Loc1 = 'Iasi' AND Loc2 = 'Focsani'
        UNION
```

```
— rute printr-o localitate intermediara
SELECT d1.Loc1, d2.Loc2,
        d1.Loc1 || '**' || d1.Loc2 || '**' || d2.Loc2,
        d1.Distanta + d2.Distanta
FROM distante_tot d1 INNER JOIN distante_tot d2
        ON d1.Loc1 = 'Iasi' AND d1.Loc2 = d2.Loc1 AND d2.Loc2 = 'Focsani'
        UNION
— rute prin doua localitati intermediare
SELECT d1.Loc1, d3.Loc2,
        d1.Loc1 || '**' || d1.Loc2 || '**' || d2.Loc2 || '**' || d3.Loc2,
        d1.Distanta + d2.Distanta + d3.Distanta
FROM distante_tot d1
        INNER JOIN distante_tot d2 ON d1.Loc1 = 'Iasi' AND d1.Loc2 = d2.Loc1
                AND d2.Loc2 <> d1.Loc1 AND d2.Loc2<>d1.Loc2
        INNER JOIN distante_tot d3 ON d2.Loc2 = d3.Loc1 AND d3.Loc2 = 'Focsani'
                AND d3.Loc2 <> d2.Loc1 AND d3.Loc2<>d2.Loc2
                AND d3.Loc2 <> d1.Loc1 AND d3.Loc2<>d1.Loc2
        UNION
— rute prin trei localitati intermediare
SELECT d1.loc1, d4.Loc2,
        d1.loc1 || '**' || d1.Loc2 || '**' || d2.Loc2 || '**' || d3.Loc2 || '**' || d4.Loc2,
        d1.distanta + d2.distanta + d3.distanta + d4.distanta
FROM distante_tot d1
        INNER JOIN distante_tot d2 ON d1.Loc1 = 'Iasi' AND d1.Loc2 = d2.Loc1
                AND d2.Loc2 <> d1.loc1 AND d2.Loc2<>d1.Loc2
        INNER JOIN distante_tot d3 ON d2.Loc2 = d3.Loc1
                AND d3.Loc2 <> d2.loc1 AND d3.Loc2<>d2.loc2
                AND d3.Loc2 <> d1.Loc1 AND d3.Loc2<>d1.Loc2
        INNER JOIN distante_tot d4 ON d3.Loc2 = d4.Loc1 AND d4.Loc2 = 'Focsani'
                AND d4.Loc2 <> d3.loc1 AND d4.Loc2<>d3.loc2
                AND d4.Loc2 <> d2.Loc1 AND d4.Loc2<>d2.Loc2
                AND d4.Loc2 <> d1.Loc1 AND d4.Loc2<>d1.Loc2
UNION
— rute prin patru localitati intermediare
SELECT d1.loc1, d5.Loc2,
        d1.loc1 || '**' || d1.Loc2 || '**' || d2.Loc2 || '**' || d3.Loc2 || '**' || d4.Loc2
                || '**' || d5.Loc2,
        d1.distanta + d2.distanta + d3.distanta + d4.distanta + d5.distanta
FROM distante_tot d1
        INNER JOIN distante_tot d2 ON d1.Loc1 = 'Iasi' AND d1.Loc2 = d2.Loc1
```

```
            AND d2.Loc2 <> d1.loc1 AND d2.Loc2<>d1.Loc2
    INNER JOIN distante_tot d3 ON d2.Loc2 = d3.Loc1
            AND d3.Loc2 <> d2.loc1 AND d3.Loc2<>d2.loc2
            AND d3.Loc2 <> d1.Loc1 AND d3.Loc2<>d1.Loc2
    INNER JOIN distante_tot d4 ON d3.Loc2 = d4.Loc1
            AND d4.Loc2 <> d3.loc1 AND d4.Loc2<>d3.loc2
            AND d4.Loc2 <> d2.Loc1 AND d4.Loc2<>d2.Loc2
            AND d4.Loc2 <> d1.Loc1 AND d4.Loc2<>d1.Loc2
    INNER JOIN distante_tot d5 ON d4.Loc2 = d5.Loc1 AND d5.Loc2 = 'Focsani'
            AND d5.Loc2 <> d4.loc1 AND d5.Loc2<>d4.loc2
            AND d5.Loc2 <> d3.loc1 AND d5.Loc2<>d3.loc2
            AND d5.Loc2 <> d2.Loc1 AND d5.Loc2<>d2.Loc2
            AND d5.Loc2 <> d1.Loc1 AND d5.Loc2<>d1.Loc2
    UNION
-- rute prin cinci localitati intermediare
SELECT d1.loc1, d6.Loc2,
    d1.loc1 || '**' || d1.Loc2 || '**' || d2.Loc2 || '**' || d3.Loc2 || '**' || d4.Loc2
            || '**' || d5.Loc2 || '**' || d6.Loc2,
    d1.distanta + d2.distanta + d3.distanta + d4.distanta + d5.distanta + d6.distanta
FROM distante_tot d1
    INNER JOIN distante_tot d2 ON d1.Loc1 = 'Iasi'
            AND d1.Loc2 = d2.Loc1 AND d2.Loc2 <> d1.loc1 AND d2.Loc2<>d1.Loc2
    INNER JOIN distante_tot d3 ON d2.Loc2 = d3.Loc1 AND d3.Loc2 <> d2.loc1
            AND d3.Loc2<>d2.loc2 AND d3.Loc2 <> d1.Loc1 AND d3.Loc2<>d1.Loc2
    INNER JOIN distante_tot d4 ON d3.Loc2 = d4.Loc1  AND d4.Loc2 <> d3.loc1
            AND d4.Loc2<>d3.loc2 AND d4.Loc2 <> d2.Loc1 AND d4.Loc2<>d2.Loc2
            AND d4.Loc2 <> d1.Loc1 AND d4.Loc2<>d1.Loc2
    INNER JOIN distante_tot d5 ON d4.Loc2 = d5.Loc1 AND d5.Loc2 <> d4.loc1
            AND d5.Loc2<>d4.loc2 AND d5.Loc2 <> d3.loc1 AND d5.Loc2<>d3.loc2
            AND d5.Loc2 <> d2.Loc1 AND d5.Loc2<>d2.Loc2
            AND d5.Loc2 <> d1.Loc1 AND d5.Loc2<>d1.Loc2
    INNER JOIN distante_tot d6 ON d5.Loc2 = d6.Loc1 AND d6.Loc2 = 'Focsani'
            AND d6.Loc2 <> d5.loc1 AND d6.Loc2<>d5.loc2
            AND d6.Loc2 <> d4.loc1 AND d6.Loc2<>d4.loc2
            AND d6.Loc2 <> d3.loc1 AND d6.Loc2<>d3.loc2
            AND d6.Loc2 <> d2.Loc1 AND d6.Loc2<>d2.Loc2
            AND d6.Loc2 <> d1.Loc1 AND d6.Loc2<>d1.Loc2
ORDER BY 4
```

Ceea ce obţinem seamănă cu rezultatul din figura 12.9. Deşi exagerat de lungă, soluţia poate fi aplicată pentru orice pereche de localităţi (de pornire, respectiv,

sosire). Unde mai punem la socoteală că în PostgreSQL, expresiile-tabelă nu funcționează.

PLECARE	SOSIRE	SIR	DISTANTA
Iasi	Focsani	Iasi**Vaslui**Birlad**Tecuci**Tisita**Focsani	206
Iasi	Focsani	Iasi**Tg.Frumos**Roman**Bacau**Adjud**Tisita**Focsani	239
Iasi	Focsani	Iasi**Vaslui**Bacau**Adjud**Tisita**Focsani	261
Iasi	Focsani	Iasi**Vaslui**Roman**Bacau**Adjud**Tisita**Focsani	297
Iasi	Focsani	Iasi**Vaslui**Birlad**Tecuci**Galati**Focsani	320
Iasi	Focsani	Iasi**Vaslui**Birlad**Tecuci**Galati**Braila**Focsani	337

Figura 12.9. Traseele (prin maximum cinci localități intermediare) dintre Iaşi şi Focşani

Vom vedea, însă, că există şi variante mai elegante de rezolvare a acestei probleme.

12.2. Interogări arborescente în Oracle

Pentru problemele formulate în acest paragraf, până în acest punct au fost formulate numai soluții (mai mult sau mai puțin) generale. Dialectele SQL ale celor patru servere au câteva opțiuni speciale pentru parcurgerea elegantă a structurilor ierarhice şi/sau recursive. Începem cu Oracle.

Care este nivelul ierarhic al fiecărui salariat?

În Oracle există două clauze prin care se construieşte o structură ierarhică/iterativă, START WITH şi CONNECT BY. Soluția următoare conduce la rezultatul din figura 12.10.

```
SELECT PERSONAL2.*, LEVEL
FROM personal2
START WITH MarcaSef IS NULL
CONNECT BY PRIOR Marca = MarcaSef
```

Construirea structurii ierarhice începe cu înregistrarea (înregistrările) care îndeplinesc condiția din clauza START WITH. Această înregistrare-părinte (care este „rădăcina" ierarhiei) va fi legată de înregistrarea sau înregistrările-copil prin condiția din CONNECT BY PRIOR Marca = MarcaSef. La rândul lor, înregistrările-copil pot avea înregistrări-copil („nepoții" rădăcinii) care sunt extrase prin acelaşi predicat din CONNECT BY.

Aşadar, prin CONNECT BY sunt selectate toate generațiile succesive de linii-copil (copii, nepoți, strănepoți etc.). Clauza PRIOR plasată în stânga condiției semnifică: valoarea atributului Marca din părinte trebuie să fie egală cu valoarea MarcaŞef din înregistrările-copil, nepot, strănepot, ...stră (de *n* ori) nepot.

După construirea ierarhiei, se elimină tuplurile ce nu îndeplinesc condiția formulată în clauza WHERE. Este important de notat că selecția se aplică linie cu linie, iar eliminarea unei linii-părinte nu atrage automat eliminarea copiilor, nepoților

ş.a.m.d. Dacă există clauza ORDER BY, aceasta va determina dispunerea înregistrărilor în rezultat. În lipsa clauzei de ordonare, înregistrările sunt dispuse în funcţie de ordinea parcurgerii arborelui, aşa cum se observă în figura 12.10.

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR	LEVEL
1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600	1
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450	2
4	ANGAJAT 4	(null)	FINANCIAR	2	1380	3
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420	3
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350	4
7	ANGAJAT 7	(null)	FINANCIAR	5	1280	4
3	ANGAJAT 3	02-08-1962	MARKETING	1	1450	2
8	ANGAJAT 8	31-12-1960	MARKETING	3	1290	3
9	ANGAJAT 9	28-02-1976	MARKETING	3	1410	3
10	ANGAJAT 10	29-01-1972	RESURSE UMANE	1	1370	2

Figura 12.10. Interogare ierarhică în Oracle

Un avantaj major a interogărilor ierarhice ţine de folosirea pseudocoloanei LEVEL ce semnifică tocmai nivelul ierarhiei, raportat la înregistrarea/înregistrările-rădăcină care îndeplineşte/îndeplinesc condiţia din START WITH. Dintre principalele restricţii Oracle trebuie amintit că SELECT-ul care execută o interogare ierarhică nu poate efectua o joncţiune şi nici nu poate extrage date dintr-o tabelă virtuală creată printr-o joncţiune.

Dacă în clauza CONNECT BY PRIOR inversăm ordinea atributelor, rezultatul va conţine o singură linie (vezi figura 12.11), întrucât linia de pornire este a Angajatului 1 pentru care nu există nici o linie-părinte (angajat superior ierarhic) pentru care marca să fie egală cu marca Angajatului 1.

```
SELECT personal2.*, LEVEL
FROM personal2
START WITH MarcaSef IS NULL
CONNECT BY PRIOR MarcaSef = Marca
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR	LEVEL
1	1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600	1

Figura 12.11. Inversarea ordinii atributelor în predicatul clauzei CONNECT BY PRIOR

Explicaţiile de mai sus sunt corecte doar pe jumătate, deoarece o ierarhie poate fi construită şi de sus în jos (TOP-DOWN) şi de jos în sus (BOTTOM-UP). În prima interogare, am specificat înregistrarea-rădăcină şi apoi condiţia prin care sunt extraşi copii, nepoţii, strănepoţii etc. Dacă am dori să construim structura de sus în jos, începând de la Angajat 2, interogarea ar fi:

SELECT personal2.*, LEVEL

FROM personal2

START WITH NumePren = 'ANGAJAT 2'

CONNECT BY PRIOR Marca = MarcaSef

iar rezultatul cel din figura 12.12. Titulaturile *părinte-copii-nepoţi-strănepoţi...* sunt, deci, corecte.

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR	LEVEL
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450	1
4	ANGAJAT 4	(null)	FINANCIAR	2	1380	2
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420	2
6	ANGAJAT 6	09-11-1965	FINANCIAR	5	1350	3
7	ANGAJAT 7	(null)	FINANCIAR	5	1280	3

Figura 12.12. Construirea ierarhiei de sus în jos

Ierarhiile pot fi însă construite şi de jos în sus (BOTTOM-UP), specificând „cea mai de jos" înregistrare şi găsind apoi, rând pe rând, superiorii ierarhici de ordin 1, 2 etc. În aceste situaţii, titulaturile *părinte-copil-nepot...* nu mai sunt operaţionale, pentru că se „urcă pe linie genealogică". Interogarea următoare, ce obţine lista din figura 12.13, extrage toţi superiorii Angajatului 7:

SELECT personal2.*, LEVEL

FROM personal2

START WITH NumePren = 'ANGAJAT 7'

CONNECT BY PRIOR MarcaSef = Marca

MARCA	NUMEPREN	DATANAST	COMPART	MARCASEF	SALTARIFAR	LEVEL
7	ANGAJAT 7	(null)	FINANCIAR	5	1280	1
5	ANGAJAT 5	30-04-1965	FINANCIAR	2	1420	2
2	ANGAJAT 2	11-10-1977	FINANCIAR	1	1450	3
1	ANGAJAT 1	01-07-1962	DIRECTIUNE	(null)	1600	4

Figura 12.13. Construirea ierarhiei de jos în sus

Aşa că este mai nimerit să folosim termenii *subordonaţi de ordinul (nivelul) 1, 2, ..., n*, respectiv *superiori de ordinul 1, 2, ..., n*. Acum că tot am lămurit modalităţile de construire a ierarhiilor, putem rezolva destul de simplu problemele formulate în paragraful anterior.

Cum se numeşte şeful Angajatului 7?

Una dintre variantele generalizabile ţine de includerea ultimei interogări BOTTOM-UP într-o expresie-tabelă şi extragerea liniei pentru care nivelul (LEVEL) angajatului este superior cu 1:

```
WITH sefi_angajat7 AS
    (SELECT personal2.*, LEVEL AS Nivel
    FROM personal2
    START WITH NumePren = 'ANGAJAT 7'
    CONNECT BY PRIOR MarcaSef = Marca )
```

```
SELECT *
FROM sefi_angajat7
WHERE Nivel = (SELECT Nivel + 1
        FROM sefi_angajat7
        WHERE NumePren = 'ANGAJAT 7')
```

Ierarhia se construieşte începând cu Angajat 7, care va avea implicit nivelul (LEVEL) 1. Este sigur, deci, că şeful direct al acestuia va avea nivelul 2, aşa că putem simplifica interogarea în oarecare măsură:

```
WITH sefi_angajat7 AS
        (SELECT personal2.*, LEVEL AS Nivel
         FROM personal2
         START WITH NumePren = 'ANGAJAT 7'
        CONNECT BY PRIOR MarcaSef = Marca
        )
SELECT * FROM sefi_angajat7 WHERE Nivel = 2
```

Care sunt subordonații direcți ai Angajatului 2?

Am răspuns deja la această întrebare când am construit ierarhia de sus în jos prin interogarea al cărei rezultat era cel din figura 12.13.

Care sunt subordonații subordonaților directorului general?

Ierarhia se construieşte de sus în jos, începând cu directorul general. Condiția poate fi exprimată astfel:

```
WITH ierarhie AS (SELECT PERSONAL2.*, LEVEL AS Nivel
                FROM personal2
                START WITH MarcaSef IS NULL
                CONNECT BY PRIOR Marca = MarcaSef)
SELECT * FROM ierarhie
WHERE Nivel = (SELECT Nivel + 2 FROM ierarhie WHERE MarcaSef IS NULL)
```

sau, dacă ținem seama că nivelul ierarhic al directorului general este 1, al subordonaților săi direcți este 2, iar al subordonaților subordonaților este 3:

```
SELECT personal2.*, LEVEL
FROM personal2
WHERE LEVEL = 3
START WITH MarcaSef IS NULL CONNECT BY MarcaSef = PRIOR Marca
```

Câte nivele ierarhice are firma?

Trebuie să aflăm LEVEL-ul maxim, ceea ce nu e chiar complicat:

```
SELECT MAX(Nivel)
FROM (SELECT personal2.*, LEVEL AS Nivel
        FROM personal2
        START WITH MarcaSef IS NULL
```

```
        CONNECT BY MarcaSef = PRIOR Marca )
```

sau, şi mai simplu:

```
SELECT MAX(LEVEL)
FROM personal2
START WITH MarcaSef IS NULL CONNECT BY MarcaSef = PRIOR Marca
```

Să se afişeze structura ierarhică a firmei.

Nu putem să reprezentăm arborele ierarhic cu „verdele în jos", ci cu rădăcina la stânga, la fel ca în soluţia cu rezultatul în figura 12.8. Indentarea subordonaţilor se obţine cu ajutorul funcţiei LPAD şi al pseudoatributului LEVEL după cum urmează:

```
SELECT LPAD(' ', 3 * (LEVEL - 1), '– ') || NumePren AS Nume, Compart
FROM personal2
START WITH MarcaSef IS NULL
CONNECT BY PRIOR Marca = MarcaSef
```

Pentru afişarea structurii ierarhice, inclusiv a modului de subordonare a fiecărui angajat se poate folosi în Oracle o funcţie specială – SYS_CONNECT_BY_PATH:

```
SELECT SYS_CONNECT_BY_PATH(NumePren,' -> ') AS "Cale de subordonare",
Compart
FROM personal2
START WITH MarcaSef IS NULL
CONNECT BY PRIOR Marca = MarcaSef
```

Funcţia returnează întreaga „cale" de subordonare a fiecărui angajat (vezi figura 12.14). Între nivelele ierarhice se poate insera un caracter sau un şir de caractere (în exemplul nostru este ->).

Cale de subordonare	COMPART
-> ANGAJAT 1	DIRECTIUNE
-> ANGAJAT 1 -> ANGAJAT 2	FINANCIAR
-> ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 4	FINANCIAR
-> ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5	FINANCIAR
-> ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 6	FINANCIAR
-> ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 7	FINANCIAR
-> ANGAJAT 1 -> ANGAJAT 3	MARKETING
-> ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 8	MARKETING
-> ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 9	MARKETING
-> ANGAJAT 1 -> ANGAJAT 10	RESURSE UMANE

Figura 12.14. Funcţia SYS_CONNECT_BY_PATH

În lipsa unei condiţii formulate în clauza START WITH, se construieşte câte o ierarhie pentru fiecare angajat (fiecare angajat va fi, pe rând, rădăcină a unei ierarhii). Astfel, interogarea următoare va genera rezultatul din figura 12.15.

NUME	COMPART
--> ANGAJAT 2	FINANCIAR
--> ANGAJAT 2 --> ANGAJAT 4	FINANCIAR
--> ANGAJAT 2 --> ANGAJAT 5	FINANCIAR
--> ANGAJAT 2 --> ANGAJAT 5 --> ANGAJAT 6	FINANCIAR
--> ANGAJAT 2 --> ANGAJAT 5 --> ANGAJAT 7	FINANCIAR
--> ANGAJAT 3	MARKETING
--> ANGAJAT 3 --> ANGAJAT 8	MARKETING
--> ANGAJAT 3 --> ANGAJAT 9	MARKETING
--> ANGAJAT 10	RESURSE UMANE
--> ANGAJAT 4	FINANCIAR
--> ANGAJAT 5	FINANCIAR
--> ANGAJAT 5 --> ANGAJAT 6	FINANCIAR
--> ANGAJAT 5 --> ANGAJAT 7	FINANCIAR
--> ANGAJAT 8	MARKETING
--> ANGAJAT 9	MARKETING
--> ANGAJAT 6	FINANCIAR
--> ANGAJAT 7	FINANCIAR
--> ANGAJAT 1	DIRECTIUNE
--> ANGAJAT 1 --> ANGAJAT 2	FINANCIAR
--> ANGAJAT 1 --> ANGAJAT 2 --> ANGAJAT 4	FINANCIAR
--> ANGAJAT 1 --> ANGAJAT 2 --> ANGAJAT 5	FINANCIAR
--> ANGAJAT 1 --> ANGAJAT 2 --> ANGAJAT 5 --> ANGAJAT 6	FINANCIAR
--> ANGAJAT 1 --> ANGAJAT 2 --> ANGAJAT 5 --> ANGAJAT 7	FINANCIAR
--> ANGAJAT 1 --> ANGAJAT 3	MARKETING
--> ANGAJAT 1 --> ANGAJAT 3 --> ANGAJAT 8	MARKETING
--> ANGAJAT 1 --> ANGAJAT 3 --> ANGAJAT 9	MARKETING
--> ANGAJAT 1 --> ANGAJAT 10	RESURSE UMANE

Figura 12.15. O ierarhie pentru fiecare angajat

```
SELECT SYS_CONNECT_BY_PATH(numepren, ' –> ') AS Nume, Compart
FROM personal2
CONNECT BY PRIOR Marca = MarcaSef
```

A sosit momentul să formulăm o soluţie mai elegantă pentru afişarea traseelor dintre două localităţi (Iaşi şi Focşani, în exemplul nostru), pe baza celor aflate mai sus. Începem cu o interogare „ajutătoare”:

```
SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
        LEVEL AS Nivel, SYS_CONNECT_BY_PATH (Distanta, ' + ') AS Dist,
        d.*, ROWNUM AS Ord
FROM distante d
START WITH Loc1 = 'Iasi'
CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
ORDER BY Ord
```

Localităţile prin care trece ruta sunt separate prin două asteriscuri, iar distanţele dintre cele două localităţi prin semnul plus (vezi figura 12.16). Prin clauza LEVEL

< 20, sunt extrase numai eventualele trasee care trec prin maximum 20 de localităţi (21, cu localitatea de plecare), valorile atributului Nivel indicând destul de bine numărul de localităţi prin care trece traseul (cu excepţia plecării).

TRASEU	NIVEL	DIST	LOC1	LOC2	DISTANTA	ORD
IasiTg.Frumos	1	+53	Iasi	Tg.Frumos	53	1
IasiTg.Frumos**Roman	2	+53+40	Tg.Frumos	Roman	40	2
IasiTg.Frumos**Roman**Bacau	3	+53+40+41	Roman	Bacau	41	3
IasiTg.Frumos**Roman**Bacau**Adjud	4	+53+40+41+60	Bacau	Adjud	60	4
IasiTg.Frumos**Roman**Bacau**Adjud**Tisita	5	+53+40+41+60+32	Adjud	Tisita	32	5
IasiTg.Frumos**Roman**Bacau**Adjud**Tisita**Focsani	6	+53+40+41+60+32+13	Tisita	Focsani	13	6
IasiTg.Frumos**Roman**Bacau**Vaslui	4	+53+40+41+85	Bacau	Vaslui	85	7
IasiTg.Frumos**Roman**Bacau**Vaslui**Birlad	5	+53+40+41+85+54	Vaslui	Birlad	54	8
IasiTg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci	6	+53+40+41+85+54+48	Birlad	Tecuci	48	9
IasiTg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Galati	7	+53+40+41+85+54+48+67	Tecuci	Galati	67	10
IasiTg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Tisita	7	+53+40+41+85+54+48+20	Tecuci	Tisita	20	11
IasiTg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Tisita**Focsani	8	+53+40+41+85+54+48+20+13	Tisita	Focsani	13	12
IasiTg.Frumos**Roman**Vaslui	3	+53+40+80	Roman	Vaslui	80	13
IasiTg.Frumos**Roman**Vaslui**Birlad	4	+53+40+80+54	Vaslui	Birlad	54	14

Figura 12.16. Trasee, trasee... (fragment)

Din păcate, nu avem o funcţie „directă" pentru evaluarea adunării kilometrilor separaţi prin + (valorile atributului Dist). Am putea crea o funcţie-utilizator, dar vom fi în stare de aşa ceva abia în capitolul 16. Aşa că, pentru a calcula lungimea fiecărui traseu şi subtraseu, vom avea nevoie de o subconsultare scalară (în clauza SELECT). Pentru a urmări mai uşor logica interogării vom recurge şi la o expresie-tabelă, iar (sub)traseele se vor „opri" la Focşani:

```
WITH tab AS
        (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel,
                SYS_CONNECT_BY_PATH (Distanta, ' + ') AS Dist,
                d.*,
                ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 10
        ORDER BY Ord)
SELECT t1.*, (SELECT SUM(Distanta) FROM tab t2 WHERE t1.Traseu
                LIKE Traseu || '%') AS Km
FROM tab t1
```

Rezultatul este cel din figura 12.17. Remarcaţi opţiunea LIKE din clauza WHERE a subconsultării ce calculează numărul de kilometri ai traseului.

TRASEU	NIVEL	DIST	LOC1	LOC2	DISTANTA	ORD	KM
IasiVaslui	1	+71	Iasi	Vaslui	71	1	71
IasiVaslui**Birlad	2	+71+54	Vaslui	Birlad	54	2	125
IasiVaslui**Birlad**Tecuci	3	+71+54+48	Birlad	Tecuci	48	3	173
IasiVaslui**Birlad**Tecuci**Tisita	4	+71+54+48+20	Tecuci	Tisita	20	4	193
IasiVaslui**Birlad**Tecuci**Tisita**Focsani	5	+71+54+48+20+13	Tisita	Focsani	13	5	206
IasiVaslui**Birlad**Tecuci**Galati	4	+71+54+48+67	Tecuci	Galati	67	6	240
IasiTg.Frumos	1	+53	Iasi	Tg.Frumos	53	7	53
IasiTg.Frumos**Roman	2	+53+40	Tg.Frumos	Roman	40	8	93
IasiTg.Frumos**Roman**Bacau	3	+53+40+41	Roman	Bacau	41	9	134
IasiTg.Frumos**Roman**Bacau**Adjud	4	+53+40+41+60	Bacau	Adjud	60	10	194
IasiTg.Frumos**Roman**Bacau**Adjud**Tisita	5	+53+40+41+60+32	Adjud	Tisita	32	11	226

Figura 12.17. Trasee şi subtrasee, cu plecare din Iaşi, şi destinaţie Focşani (fragment)

Dacă dorim doar afişarea „rutelor întregi" Iaşi-Focşani, modificăm uşor interogarea de mai sus (vezi figura 12.18):

```
WITH tab AS (
    SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
        LEVEL AS Nivel, d.*, ROWNUM AS Ord
    FROM distante d
    START WITH Loc1 = 'Iasi'
    CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 50
    ORDER BY Ord)
SELECT Ord,Traseu, Dist, (SELECT SUM(Distanta) FROM tab t2
        WHERE t1.Traseu LIKE Traseu || '%') AS Km
FROM tab t1
WHERE Traseu LIKE '**Iasi%Focsani'
ORDER BY Km
```

ORD	TRASEU	DIST	KM
5	**Iasi**Vaslui**Birlad**Tecuci**Tisita**Focsani	+71+54+48+20+13	206
12	**Iasi**Tg.Frumos**Roman**Bacau**Adjud**Tisita**Focsani	+53+40+41+60+32+13	239
23	**Iasi**Tg.Frumos**Roman**Vaslui**Birlad**Tecuci**Tisita**Focsani	+53+40+80+54+48+20+13	308
17	**Iasi**Tg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Tisita**Focsani	+53+40+41+85+54+48+20+13	354

Figura 12.18. Trasee întregi Iaşi-Focşani

În continuare, ne punem problema generării staţiilor pentru o rută, „lung prilej de bârfe şi de ipoteze". Iată o soluţie care impresionează prin dimensiune:

```
SELECT rute.*,
    SUBSTR(traseu, INSTR(traseu,'**',1,1) + 2, INSTR(traseu,'**',1,2) -3) AS L1,
    CASE
    WHEN INSTR(traseu,'**',1,2) = 0 THEN NULL
    ELSE SUBSTR(traseu, INSTR(traseu,'**',1,2) + 2,
        CASE
```

```
                WHEN INSTR(traseu, '**',1,3) > 0 THEN INSTR(traseu,'**',1,3) -
                        INSTR(traseu,'**',1,2) - 2
                ELSE LENGTH(traseu)
                END )
        END AS L2,
        CASE
        WHEN INSTR(traseu,'**',1,3) = 0 THEN NULL
        ELSE SUBSTR(traseu, INSTR(traseu,'**',1,3) + 2,
                CASE
                WHEN INSTR(traseu, '**',1,4) > 0 THEN INSTR(traseu,'**',1,4) -
                        INSTR(traseu,'**',1,3) - 2
                ELSE LENGTH(traseu)
                END)
        END AS L3,
        CASE
        WHEN INSTR(traseu,'**',1,4) = 0 THEN NULL
        ELSE SUBSTR(traseu, INSTR(traseu,'**',1,4) + 2,
                CASE
                WHEN INSTR(traseu, '**',1,5) > 0 THEN INSTR(traseu,'**',1,5) -
                        INSTR(traseu,'**',1,4) - 2
                ELSE LENGTH(traseu)
                END)
        END AS L4,
        CASE
        WHEN INSTR(traseu,'**',1,5) = 0 THEN NULL
        ELSE SUBSTR(traseu, INSTR(traseu,'**',1,5) + 2,
                CASE
                WHEN INSTR(traseu, '**',1,6) > 0 THEN INSTR(traseu,'**',1,6) -
                        INSTR(traseu,'**',1,5) - 2
                ELSE LENGTH(traseu)
                END )
        END AS L5,
        CASE
        WHEN INSTR(traseu,'**',1,6) = 0 THEN NULL
        ELSE SUBSTR(traseu, INSTR(traseu,'**',1,6) + 2,
                CASE
                WHEN INSTR(traseu, '**',1,7) > 0 THEN INSTR(traseu,'**',1,7) -
                        INSTR(traseu,'**',1,6) - 2
                ELSE LENGTH(traseu)
                END)
```

```
        END AS L6,
        CASE
        WHEN INSTR(traseu,'**',1,7) = 0 THEN NULL
        ELSE SUBSTR(traseu, INSTR(traseu,'**',1,7) + 2,
                CASE
                WHEN INSTR(traseu, '**',1,8) > 0 THEN INSTR(traseu,'**',1,8) -
                        INSTR(traseu,'**',1,7) - 2
                ELSE LENGTH(traseu)
                END )
        END AS L7
FROM    (
        SELECT Ord, Traseu, (SELECT SUM(Distanta)
                        FROM (
                        SELECT SYS_CONNECT_BY_PATH (Loc1,
                                '**') || '**' || Loc2 AS Traseu,
                                LEVEL AS Nivel,
                                d.*,
                                ROWNUM AS Ord
                                FROM distante d
                                START WITH Loc1 = 'Iasi'
                                CONNECT BY PRIOR Loc2 = Loc1 AND
                                        Loc1<>'Focsani' AND Level < 50
                                ORDER BY Ord
                                        ) t2
                                WHERE tab.Traseu LIKE Traseu|| '%') AS Km
        FROM (
                SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**'
                        || Loc2 AS Traseu, LEVEL AS Nivel,
                        SYS_CONNECT_BY_PATH (Distanta, ' + ') AS Dist,
                        d.*,
                        ROWNUM AS Ord
                FROM distante d
                START WITH Loc1 = 'Iasi'
                CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND
                        Level < 50
                ORDER BY Ord
                ) tab
        WHERE Traseu LIKE '**Iasi%Focsani'
        ORDER BY Km
        ) rute
```

Chiar dacă rezultatul din figura 12.19 este încurajator, efortul de scriere este insuportabil, mai ales dacă ne gândim că e posibil să repetăm operațiunea pentru traseele cu 20 de localități sau chiar mai multe.

ORD	TRASEU	KM	L1	L2	L3	L4	L5	L6	L7
24	**Iasi**Vaslui**Birlad**Tecuci**Tisita**Focsani	206	Iasi	Vaslui	Birlad	Tecuci	Tisita	Focsani	(null)
6	**Iasi**Tg.Frumos**Roman**Bacau**Adjud**Tisita**Focsani	239	Iasi	Tg.Frumos	Roman	Bacau	Adjud	Tisita	Focsani
18	**Iasi**Tg.Frumos**Roman**Vaslui**Birlad**Tecuci**Tisita**Focsani	308	Iasi	Tg.Frumos	Roman	Vaslui	Birlad	Tecuci	Tisita
12	**Iasi**Tg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Tisita**Focsani	354	Iasi	Tg.Frumos	Roman	Bacau	Vaslui	Birlad	Tecuci

Figura 12.19. Localitățile (stațiile) de pe traseele Iaşi-Focşani – varianta incompletă

Aşa că avem motive suficiente să ne străduim în a găsi o soluție mai eficientă. Ce ziceți de următoarea prin care extragem, pe câte o linie, câte o stație (localitate) de pe traseul Iaşi-Focşani via Vaslui?

```
WITH tab AS
        (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel, d.*, ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord
        )
SELECT DISTINCT t2.traseu, t2.Nivel AS LocNr, t2.Loc2
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '**Iasi**Vaslui%Focsani' AND
                t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2
```

Să recunoaştem că soluția este de-a dreptul simpatică, prin comparație cu precedenta, iar rezultatul este corect - sau cel puțin aşa pare în figura 12.20.

TRASEU	LOCNR	LOC2
IasiVaslui	1	Vaslui
IasiVaslui**Birlad	2	Birlad
IasiVaslui**Birlad**Tecuci	3	Tecuci
IasiVaslui**Birlad**Tecuci**Tisita	4	Tisita
IasiVaslui**Birlad**Tecuci**Tisita**Focsani	5	Focsani

Figura 12.20. Localitățile (stațiile) de pe traseul Iaşi-Vaslui-Focşani

12.3. Interogări arborescente în MS SQL Server

Sistemul de construire a unei ierarhii în MS SQL Server este sensibil diferit de cel din Oracle:

```
WITH ierarhie ( Marca, NumePren, Compart, MarcaSef, Nivel) AS
        (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
```

FROM personal2

WHERE MarcaSef IS NULL

UNION ALL

SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1

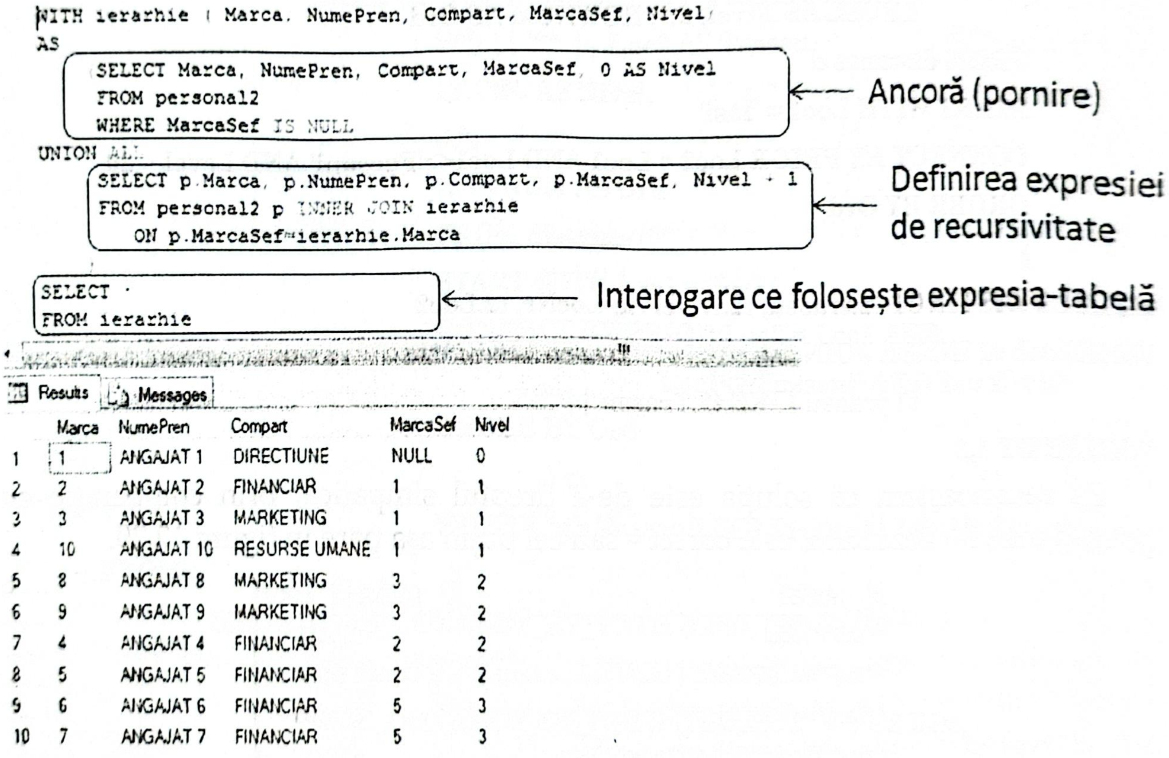
FROM personal2 p INNER JOIN ierarhie ON p.MarcaSef = ierarhie.Marca

)

SELECT *

FROM ierarhie

Mai întâi, sintaxa presupune obligatoriu folosirea unei expresii-tabelă, denumită, în cazul nostru, IERARHIE. Expresia-tabelă este definită prin două interogări conectate prin operatorul de reuniune – UNION ALL. Prima este interogarea-ancoră şi defineşte înregistrarea-rădăcină (vezi figura 12.21). A doua foloseşte ancora care este joncţionată cu tabela PERSONAL2. Expresia de joncţiune exprimă, de fapt, recursivitatea prin care se construieşte ierarhia.



	Marca	NumePren	Compart	MarcaSef	Nivel
1	1	ANGAJAT 1	DIRECTIUNE	NULL	0
2	2	ANGAJAT 2	FINANCIAR	1	1
3	3	ANGAJAT 3	MARKETING	1	1
4	10	ANGAJAT 10	RESURSE UMANE	1	1
5	8	ANGAJAT 8	MARKETING	3	2
6	9	ANGAJAT 9	MARKETING	3	2
7	4	ANGAJAT 4	FINANCIAR	2	2
8	5	ANGAJAT 5	FINANCIAR	2	2
9	6	ANGAJAT 6	FINANCIAR	5	3
10	7	ANGAJAT 7	FINANCIAR	5	3

Figura 12.21. Construirea ierarhiei în MS SQL Server

Cum se numeşte şeful Angajatului 7?

Ierarhia se va construi de jos în sus, începând cu Angajat 7, care va avea implicit nivelul (LEVEL) 0; de aceea, vom modifica predicatul din expresia de recursivitate. Şeful direct al Angajatului 7 va avea nivelul 1 (vezi figura 12.22):

WITH ierarhie7 (Marca, NumePren, Compart, MarcaSef, Nivel) AS

(SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel

FROM personal2

WHERE NumePren = 'ANGAJAT 7'

```
            UNION ALL
        SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
        FROM personal2 p INNER JOIN ierarhie7 ON p.Marca = ierarhie7.MarcaSef)
SELECT *
FROM ierarhie7
WHERE Nivel = 1
```

Marca	NumePren	Compart	MarcaSef	Nivel
5	ANGAJAT 5	FINANCIAR	2	1

Figura 12.22. Şeful direct al Angajatului 7 (MS SQL Server)

Care sunt subordonaţii direcţi ai Angajatului 2?

Modificăm uşor SELECT-ul principal al interogării din deschiderea acestui paragraf:

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
            (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
            FROM personal2
            WHERE NumePren = 'ANGAJAT 2'
        UNION ALL
            SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
            FROM personal2 p INNER JOIN ierarhie
        ON p.MarcaSef = ierarhie.Marca)
SELECT *
FROM ierarhie
WHERE Nivel = 1
```

Care sunt subordonaţii subordonaţilor directorului general?

Ierarhia de construieşte de sus în jos, începând cu directorul general, iar angajaţii care ne interesează se plasează pe nivelul 2:

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
        (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
        FROM personal2
        WHERE MarcaSef IS NULL
            UNION ALL
        SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
        FROM personal2 p INNER JOIN ierarhie ON p.MarcaSef = ierarhie.Marca)
SELECT *
FROM ierarhie
WHERE Nivel = 2
```

Câte nivele ierarhice are firma?

Ţinând seama de faptul că nivelul porneşte de la zero, trebuie să incrementăm cu 1 rezultatul obţinut prin funcţia MAX:

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
        (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
        FROM personal2
        WHERE MarcaSef IS NULL
                UNION ALL
        SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
        FROM personal2 p
                INNER JOIN ierarhie ON p.MarcaSef = ierarhie.Marca)
SELECT MAX(Nivel) + 1
FROM ierarhie
```

Să se afişeze structura ierarhică a firmei.

MS SQL Server nu prezintă o funcţie similară SYS_CONNECT_BY_PATH. Cu toate acestea, putem să furnizăm un rezultat de forma celui din figura 12.14, inclusiv „calea de subordonare" ierarhică a fiecărui angajat. Ideea ar fi ca, în interogarea destinată precizării expresiei de recursivitate, numele şi prenumele să fie obţinut prin concatenarea *ierarhie.NumePren* + ' -> ' + *p.NumePren.* Din păcate, SQL Serverul spune că tipul lui NumePren din „ancoră" diferă de tipul obţinut prin concatenarea din propoziţia anterioară, chiar dacă toate valorile sunt şiruri de caractere (vezi figura 12.23).

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
     (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
      FROM personal2
      WHERE MarcaSef IS NULL
UNION ALL
      SELECT p.Marca, ierarhie.NumePren + ' -> ' + p.NumePren,
         p.Compart, p.MarcaSef, Nivel + 1
      FROM personal2 p INNER JOIN ierarhie
ON p.MarcaSef=ierarhie.Marca

SELECT *
FROM ierarhie

Messages
Msg 240, Level 16, State 1, Line 1
Types don't match between the anchor and the recursive part in column "NumePren"
 of recursive query "ierarhie".
```

Figura 12.23. Un naz de-al SQL Server-ului

Rezolvarea vine de la funcţia CAST:

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
```

```
        (SELECT Marca, CAST(NumePren AS VARCHAR(500)),
                Compart, MarcaSef, 0 AS Nivel
        FROM personal2
        WHERE MarcaSef IS NULL
                UNION ALL
        SELECT p.Marca, CAST (ierarhie.NumePren + ' -> '
                + p.NumePren AS VARCHAR(500)), p.Compart, p.MarcaSef, Nivel + 1
        FROM personal2 p INNER JOIN ierarhie ON p.MarcaSef = ierarhie.Marca)
SELECT *
FROM ierarhie
ORDER BY 2
```

De data aceasta, SQL Server-ul este chiar cooperant, afişând rezultatul din figura 12.24.

Marca	NumePren	Compart	MarcaSef	Nivel
1	ANGAJAT 1	DIRECTIUNE	NULL	0
10	ANGAJAT 1 -> ANGAJAT 10	RESURSE UMANE	1	1
2	ANGAJAT 1 -> ANGAJAT 2	FINANCIAR	1	1
4	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 4	FINANCIAR	2	2
5	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5	FINANCIAR	2	2
6	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 6	FINANCIAR	5	3
7	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 7	FINANCIAR	5	3
3	ANGAJAT 1 -> ANGAJAT 3	MARKETING	1	1
8	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 8	MARKETING	3	2
9	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 9	MARKETING	3	2

Figura 12.24. Calea ierarhică completă a fiecărui angajat (în SQL Server)

Ca şi în Oracle, în lipsa unei condiţii formulate în interogarea-ancoră, se construieşte câte o ierarhie pentru fiecare angajat (fiecare angajat va fi, pe rând, rădăcină a unei ierarhii), astfel că interogarea:

```
WITH ierarhie ( Marca, NumePren, Compart, MarcaSef, Nivel) AS
        (SELECT Marca, CAST(NumePren AS VARCHAR(500)), Compart,
                MarcaSef, 0 AS Nivel
        FROM personal2
                UNION ALL
        SELECT p.Marca, CAST (ierarhie.NumePren + ' -> '
                + p.NumePren AS VARCHAR(500)), p.Compart, p.MarcaSef, Nivel + 1
        FROM personal2 p INNER JOIN ierarhie ON p.MarcaSef = ierarhie.Marca
        )
SELECT * FROM ierarhie ORDER BY 2
```

va genera rezultatul din figura 12.25.

Marca	NumePren	Compart	MarcaSef	Nivel
1	ANGAJAT 1	DIRECTIUNE	NULL	0
10	ANGAJAT 1 -> ANGAJAT 10	RESURSE UMANE	1	1
2	ANGAJAT 1 -> ANGAJAT 2	FINANCIAR	1	1
4	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 4	FINANCIAR	2	2
5	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5	FINANCIAR	2	2
6	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 6	FINANCIAR	5	3
7	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 7	FINANCIAR	5	3
3	ANGAJAT 1 -> ANGAJAT 3	MARKETING	1	1
8	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 8	MARKETING	3	2
9	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 9	MARKETING	3	2
10	ANGAJAT 10	RESURSE UMANE	1	0
2	ANGAJAT 2	FINANCIAR	1	0
4	ANGAJAT 2 -> ANGAJAT 4	FINANCIAR	2	1
5	ANGAJAT 2 -> ANGAJAT 5	FINANCIAR	2	1
6	ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 6	FINANCIAR	5	2
7	ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 7	FINANCIAR	5	2
3	ANGAJAT 3	MARKETING	1	0
8	ANGAJAT 3 -> ANGAJAT 8	MARKETING	3	1
9	ANGAJAT 3 -> ANGAJAT 9	MARKETING	3	1
4	ANGAJAT 4	FINANCIAR	2	0
5	ANGAJAT 5	FINANCIAR	2	0
6	ANGAJAT 5 -> ANGAJAT 6	FINANCIAR	5	1
7	ANGAJAT 5 -> ANGAJAT 7	FINANCIAR	5	1
6	ANGAJAT 6	FINANCIAR	5	0
7	ANGAJAT 7	FINANCIAR	5	0
8	ANGAJAT 8	MARKETING	3	0
9	ANGAJAT 9	MARKETING	3	0

Figura 12.25. Calea ierarhică completă a fiecărui angajat (în SQL Server)

Atacăm problema traseelor posibile dintre două localități. Construirea grafului nu ridică probleme deosebite. Astfel, interogarea:

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS
        (SELECT CAST(Loc1 + ' -> ' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Distanta AS NUMERIC), Loc2
         FROM distante WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + ' -> ' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Distanta + d.Distanta AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20
        )
SELECT * FROM ierarhie
```

obține un rezultat de forma celui din figura 12.26.

Cale	Nivel	Distanta	Loc2
Iasi -> Tg.Frumos	0	53	Tg.Frumos
Iasi -> Vaslui	0	71	Vaslui
Iasi -> Vaslui -> Birlad	1	125	Birlad
Iasi -> Vaslui -> Birlad -> Tecuci	2	173	Tecuci
Iasi -> Vaslui -> Birlad -> Tecuci -> Galati	3	240	Galati
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita	3	193	Tisita
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani	4	206	Focsani
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Braila	5	292	Braila
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Galati	5	286	Galati
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Rm.Sarat	5	248	Rm.Sarat
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Rm.Sarat -> Braila	6	332	Braila
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Rm.Sarat -> Braila -> Galati	7	343	Galati
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani -> Braila -> Galati	6	303	Galati
Iasi -> Tg.Frumos -> Roman	1	93	Roman
Iasi -> Tg.Frumos -> Roman -> Bacau	2	134	Bacau
Iasi -> Tg.Frumos -> Roman -> Vaslui	2	173	Vaslui
Iasi -> Tg.Frumos -> Roman -> Vaslui -> Birlad	3	227	Birlad
Iasi -> Tg.Frumos -> Roman -> Vaslui -> Birlad -> Tecuci	4	275	Tecuci

Figura 12.26. Trasee care încep din Iaşi (în SQL Server) – fragment

Răspunsul „punctual" la problema afişării rutelor posibile dintre Iaşi şi Focşani, în ordinea distanţelor (figura 12.27) se află acum destul de uşor:

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS
        (SELECT CAST(Loc1 + ' -> ' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Distanta AS NUMERIC), Loc2
         FROM distante
         WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + ' -> ' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Distanta + d.Distanta AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20
)
SELECT Cale, Distanta FROM ierarhie WHERE Cale LIKE 'Iasi%Focsani'
ORDER BY Distanta
```

cale	distanta
Iasi -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani	206
Iasi -> Tg.Frumos -> Roman -> Bacau -> Adjud -> Tisita -> Focsani	239
Iasi -> Tg.Frumos -> Roman -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani	308
Iasi -> Tg.Frumos -> Roman -> Bacau -> Vaslui -> Birlad -> Tecuci -> Tisita -> Focsani	354

Figura 12.27. Trasee Iaşi-Focşani (în SQL Server)

Iar pentru a obţine staţiile (localităţile) de pe acest traseu (vezi figura 12.20), interogarea va fi:

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS(
        SELECT CAST(Loc1 + ' -> ' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Distanta AS NUMERIC), Loc2
        FROM distante
        WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + ' -> ' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Distanta + d.Distanta AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20
        )
SELECT i1.Cale, i2.Nivel + 1 AS LocNr, i2.Loc2
FROM ierarhie i1
        INNER JOIN ierarhie i2 ON i1.Cale LIKE 'Iasi -> Vaslui%Focsani'
                AND i1.Cale LIKE i2.Cale + '%'
ORDER BY 1,2
```

12.4. Interogări arborescente în DB2

Sintaxa redactării interogărilor ierarhice în DB2 seamănă binişor cu surata ei din SQL Server. La fel şi logica acestor tipuri de SELECT-uri. O diferenţă, însă, este alergia la INNER JOIN în expresia de recursivitate (vezi figura 12.28):

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS (
        SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
        FROM personal2
        WHERE MarcaSef IS NULL
                UNION ALL
        SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
        FROM ierarhie i
                INNER JOIN personal2 p ON i.Marca = p.MarcaSef)
SELECT * FROM ierarhie
```

```
WITH ierarhie ( Marca, NumePren, Compart, MarcaSef, Nivel) AS
(SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
 FROM personal2
 WHERE MarcaSef IS NULL
UNION ALL
 SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
 FROM ierarhie i INNER JOIN personal2 p ON i.Marca=p.MarcaSef
)
SELECT *
FROM ierarhie
```

```
WITH ierarhie ( Marca, NumePren, Compart, MarcaSef, Nivel) AS (SELECT Marc
a, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1 FROM ierarhie i INNER JOIN
SQL0345N  The fullselect of the recursive common table expression
"MARINFOTACHE.IERARHIE" must be the UNION of two or more fullselects and
cannot include column functions, GROUP BY clause, HAVING clause, ORDER BY
clause, or an explicit join including an ON clause.  SQLSTATE=42836
```

Figura 12.28. O alergie DB2 la clauza INNER JOIN a unei interogări ierarhice

Dacă însă înlocuim INNER JOIN-ul cu sintaxa joncţiunii din SQL-89 – FROM/WHERE, lucrurile decurg fără incidente. Astfel, rezultatul din figura 12.21 se obţine prin interogarea:

```
WITH ierarhie (Marca, NumePren, Compart, MarcaSef, Nivel) AS
        (SELECT Marca, NumePren, Compart, MarcaSef, 0 AS Nivel
        FROM personal2
        WHERE MarcaSef IS NULL
                UNION ALL
        SELECT p.Marca, p.NumePren, p.Compart, p.MarcaSef, Nivel + 1
        FROM ierarhie i, personal2 p
        WHERE i.Marca = p.MarcaSef)
SELECT * FROM ierarhie
```

Astfel, înlocuind INNER JOIN-urile cu sintaxa SQL-89 şi simbolul de concatenare (din + în | |), interogările ierarhice din MS SQL Server sunt funcţionale şi în DB2[2]. Nu toate, însă. Cea de mai jos se împotriveşte destul de ciudat (vezi figura 12.29):

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS (
        SELECT CAST(Loc1 || '->' || Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Distanta AS NUMERIC), Loc2
        FROM distante
        WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.cale || '->' || d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.distanta + d.distanta AS NUMERIC), d.loc2
```

2. La drept vorbind, se presupune că MS SQL Server-ul s-a inspirat din DB2, chiar dacă noi le-am prezentat în acest capitol în ordine cronologic inversă.

```
        FROM ierarhie i, distante d
        WHERE i.Loc2 = d.Loc1 AND Nivel < 20
        )
SELECT i1.cale, i2.nivel + 1 AS LocNr, i2.Loc2
FROM ierarhie i1, ierarhie i2
WHERE i1.Cale LIKE 'Iasi -> Vaslui%Focsani' AND i1.Cale LIKE i2.Cale || '%'
ORDER BY 1,2
```

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS(
        SELECT CAST(Loc1 || ' -> ' || Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Distanta AS NUMERIC), Loc2
        FROM distante
        WHERE Loc1='Iasi'
                UNION ALL
        SELECT CAST (i.cale || ' -> ' || d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.distanta+d.distanta AS NUMERIC), d.loc2
        FROM ierarhie i, distante d
        WHERE i.Loc2=d.Loc1 AND Nivel < 20
        )
SELECT i1.cale, i2.nivel + 1 AS LocNr, i2.Loc2
FROM ierarhie i1, ierarhie i2
WHERE i1.Cale LIKE 'Iasi -> Vaslui%Focsani' AND i1.Cale LIKE i2.Cale || '%'
ORDER BY 1,2
```

```
SQL0132N  A LIKE predicate or POSSTR scalar function is not valid because the
first operand is not a string expression or the second operand is not a
string.  A LOCATE or POSITION scalar function is not valid because the first
operand is not a string or the second operand is not a string expression.
SQLSTATE=42824
```

Figura 12.29. O altă alergie DB2 (la cea de-a doua clauză LIKE)

Ştim măcar de unde ni se trage. Noroc că, ultimul LIKE se poate substitui cu o funcţie SUBSTR, astfel că problema e rezolvată:

```
WITH ierarhie (Cale, Nivel, Distanta, Loc2) AS (
        )
SELECT i1.cale, i2.nivel + 1 AS LocNr, i2.Loc2
FROM ierarhie i1, ierarhie i2
WHERE i1.Cale LIKE 'Iasi -> Vaslui%Focsani' AND
        SUBSTR(i1.Cale,1,LENGTH(i2.Cale)) = i2.Cale
ORDER BY 1,2
```

12.5. Interogări arborescente în PostgreSQL

În materie de opțiuni ierarhice şi recursive, PostgreSQL-ul este mult mai modest decât greii bazelor de date[3]. Doar interogările clasice din primul paragraf (făcând abstracție de inexistența expresiilor-tabelă WITH ... SELECT ...) sunt funcționale. Singura speranță este o funcție-tabelă (ce returnează un set de înregistrări) numită CONNECTBY. Sintaxa sa este destul de laborioasă[4]. Astfel, interogarea:

```
SELECT *
FROM CONNECTBY('personal2', 'Marca', 'MarcaSef', '1', 0,'->')
        AS t (Marca NUMERIC, MarcaSef NUMERIC, LEVEL INT, cale text)
```

obține liniile din figura 12.30. Ordinea argumentelor trebuie nimerită (din una sau mai multe încercări).

marca numeric	marcasef numeric	level integer	cale text
1		0	1
2	1	1	1->2
4	2	2	1->2->4
5	2	2	1->2->5
6	5	3	1->2->5->6
7	5	3	1->2->5->7
3	1	1	1->3
8	3	2	1->3->8
9	3	2	1->3->9
10	1	1	1->10

Figura 12.30. Funcția CONNECTBY din PostgreSQL

Rezultatul unei funcții CONNECTBY este o tabelă, aşa că o putem folosi după modelul subconsultărilor din clauza FROM (vezi figura 12.31):

```
SELECT *
FROM
        (SELECT *
         FROM CONNECTBY('personal2', 'Marca', 'MarcaSef', '1', 0,'->')
                AS t (Marca NUMERIC, MarcaSef NUMERIC, LEVEL INT, cale text)
        ) ierarhie
                INNER JOIN personal2 p ON ierarhie.Marca = p.Marca
```

3. În ediția din 18 mai 2008 a săptămânalului „buletin de ştiri" PostgreSQL (disponibil la adresa *http://people.planetpostgresql.org/dfetter/index.php?/archives/173-PostgreSQL-Weekly-News-May-18-2008.html* a fost anunțată comanda WITH RECURSIVE disponibilă, probabil, într-o viitoare (sub)versiune, grație unui patch al cărui autor este Tatsuoo Ishii.
4. Vezi explicațiile de la adresa: *http://www.postgresql.org/docs/8.3/interactive/tablefunc.html*.

marca numeric	marcasef numeric	level integer	cale text	marca numeric(5,0)	numepren character var	datanast date	compart character var	marcasef numeric(5,0)	saltarifar numeric(12,2
1		0	1	1	ANGAJAT 1	1962-07-01	DIRECTIUNE		1600.00
2	1	1	1->2	2	ANGAJAT 2	1977-10-11	FINANCIAR	1	1450.00
4	2	2	1->2->4	4	ANGAJAT 4		FINANCIAR	2	1380.00
5	2	2	1->2->5	5	ANGAJAT 5	1965-04-30	FINANCIAR	2	1420.00
6	5	3	1->2->5->6	6	ANGAJAT 6	1965-11-09	FINANCIAR	5	1350.00
7	5	3	1->2->5->7	7	ANGAJAT 7		FINANCIAR	5	1280.00
3	1	1	1->3	3	ANGAJAT 3	1962-08-02	MARKETING	1	1450.00
8	3	2	1->3->8	8	ANGAJAT 8	1960-12-31	MARKETING	3	1290.00
9	3	2	1->3->9	9	ANGAJAT 9	1976-02-28	MARKETING	3	1410.00
10	1	1	1->10	10	ANGAJAT 10	1972-01-29	RESURSE UMAN	1	1370.00

Figura 12.31. Joncţiunea tabelei obţinute prin CONNECTBY cu tabela PERSONAL2

În schimb afişarea „căii de subordonare" a fiecărui angajat, după modelul din figura 12.14, nu pare a fi la îndemână, întrucât sintaxa impune ca primul argument al funcţiei CONNECTBY să fie o tabelă (sau tabelă virtuală). O idee ar fi, însă, chiar dacă ne ia gura pe dinaintea capitolului 14 în care vom discuta despre tabele virtuale. Vom crea o tabelă virtuală în care structura ierarhică o vor indica nu prin perechea Marcă/MarcăŞef, ci prin perechea NumePren/NumeŞef:

```
CREATE VIEW personal2_modif01 AS (
    SELECT p1.*, p2.NumePren AS NumeSef
    FROM personal2 p1 LEFT OUTER JOIN personal2 p2 ON p1.MarcaSef = p2.Marca)
```

Această tabelă virtuală devine obiectul muncii unei funcţii CONNECTBY:

```
SELECT *
FROM CONNECTBY('personal2_modif01', 'NumePren', 'NumeSef', 'ANGAJAT 1', 0, ' -> ')
        AS t2 (NumePren TEXT, NumeSef TEXT, LEVEL INT, cale text)
```

Rezultatul este pe măsura aşteptărilor (vezi figura 12.32).

numepren text	numesef text	level integer	cale text
ANGAJAT 1		0	ANGAJAT 1
ANGAJAT 2	ANGAJAT 1	1	ANGAJAT 1 -> ANGAJAT 2
ANGAJAT 4	ANGAJAT 2	2	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 4
ANGAJAT 5	ANGAJAT 2	2	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5
ANGAJAT 6	ANGAJAT 5	3	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 6
ANGAJAT 7	ANGAJAT 5	3	ANGAJAT 1 -> ANGAJAT 2 -> ANGAJAT 5 -> ANGAJAT 7
ANGAJAT 3	ANGAJAT 1	1	ANGAJAT 1 -> ANGAJAT 3
ANGAJAT 8	ANGAJAT 3	2	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 8
ANGAJAT 9	ANGAJAT 3	2	ANGAJAT 1 -> ANGAJAT 3 -> ANGAJAT 9
ANGAJAT 10	ANGAJAT 1	1	ANGAJAT 1 -> ANGAJAT 10

Figura 12.32. „Calea" de subordonare a fiecărui angajat

Întâmplarea face ca, în PostgreSQL, problema cu distanţele să ne fie mai la îndemână decât cea cu structura ierarhică a firmei, întrucât graful se construieşte direct din Loc1 şi Loc2. Fără explicaţii preliminare, iată interogarea PostgreSQL care ne afişează rutele între Iaşi şi Focşani şi distanţele corespunzătoare fiecărei rute (vezi figura 12.33):

```
SELECT x1.cale, SUM(x2.distanta) AS Km
FROM
```

```
        (SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta, ierarhie.Cale
         FROM (SELECT *
                 FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, '**')
                      AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
                 ) ierarhie, distante d
        WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2
        ) x1,
        (SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta, ierarhie.Cale
        FROM (SELECT *
                 FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, '**')
                      AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
                 ) ierarhie, distante d
        WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2
        ) x2
WHERE x1.cale LIKE 'Iasi%Focsani' AND x1.cale LIKE x2.cale || '%'
GROUP BY x1.cale
ORDER BY Km
```

cale text	km numeric
Iasi**Vaslui**Birlad**Tecuci**Tisita**Focsani	206
Iasi**Tg.Frumos**Roman**Bacau**Adjud**Tisita**Focsani	239
Iasi**Tg.Frumos**Roman**Vaslui**Birlad**Tecuci**Tisita**Focsani	308
Iasi**Tg.Frumos**Roman**Bacau**Vaslui**Birlad**Tecuci**Tisita**Focsani	354

Figura 12.33. Trasee şi distanţe în kilometri între Iaşi şi Focşani – varianta PostgreSQL

În fine, localităţile (staţiile) de pe traseul Iaşi-Vaslui-...Focşani (figura 12.20) se obţin prin:

```
SELECT x1.cale, x2.NiveL AS LocNr, x2.Loc2
FROM
        (SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta,
                ierarhie.Cale, LEVEL AS Nivel
        FROM (SELECT *
                FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, '**')
                        AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
                ) ierarhie, distante d
        WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2
                ) x1,
        (SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta,
                ierarhie.Cale, LEVEL AS Nivel
        FROM (SELECT *
                FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, '**')
```

```
                              AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
                    ) ierarhie, distante d
          WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2
                    ) x2
WHERE x1.cale LIKE 'Iasi**Vaslui%Focsani' AND x1.cale LIKE x2.cale || '%'
```

12.6. Interogări mai puţin arborescente, dar tot recursive

Pentru fiecare factură, cu titlu informativ, vrem să afişăm sub forma unui şir de caractere destul de lung, lista produselor din acea factură, inclusiv cantitatea vândută şi preţul unitar (consemnate în factura respectivă). Să începem cu o variantă ajutătoare. Interogarea Oracle următoare produce rezultatul din figura 12.34:

```
SELECT lf.*, SYS_CONNECT_BY_PATH(lf.NrFact || '-' || lf.Linie || '-' || DenPr, ' \ ')
          AS Liniarizare, LEVEL
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact = 1111
START WITH Linie = 1
CONNECT BY PRIOR NrFact = NrFact AND PRIOR Linie = Linie-1
ORDER BY 1, 2
```

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	LINIARIZARE	LEVEL
1111	1	1	50	1000	\ 1111-1-Produs 1	1
1111	2	2	75	1050	\ 1111-1-Produs 1 \ 1111-2-Produs 2	2
1111	3	5	500	7060	\ 1111-1-Produs 1 \ 1111-2-Produs 2 \ 1111-3-Produs 5	3

Figura 12.34. Lista produselor din factura 1111

La fiecare linie se adaugă în listă (atributul Liniarizare): numărul facturii, numărul liniei şi denumirea produsului, astfel că în rândul corespunzător liniei finale din factură avem şirul de caractere care ne interesează. Varianta SQL Server a acestei interogări Oracle este:

```
WITH ierarhie (NrFact, Linie, CodPr, Cantitate, PretUnit, Nivel, DenPr) AS
          (SELECT NrFact, Linie, lf.CodPr, Cantitate, PretUnit, 0 AS Nivel,
                    CAST ('\' + CAST (NrFact AS VARCHAR) + '-' +
                    CAST (Linie AS VARCHAR) + '-' + DenPr AS VARCHAR(500))
          FROM liniifact lf
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
          WHERE NrFact = 1111 AND Linie = 1
                    UNION ALL
          SELECT lf.NrFact, lf.Linie, lf.CodPr, lf.Cantitate, lf.PretUnit, Nivel + 1,
                    CAST (ierarhie.DenPr + ' \ ' + + CAST (lf.NrFact AS VARCHAR) +
                    '-' + CAST (lf.Linie AS VARCHAR) + '-' + p.DenPr AS VARCHAR(500))
```

```
        FROM liniifact lf
                INNER JOIN ierarhie ON lf.NrFact = ierarhie.NrFact AND
                        lf.Linie = ierarhie.Linie + 1
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE lf.NrFact = 1111
        )
SELECT * FROM ierarhie
```

iar cea DB2 este:

```
WITH ierarhie (NrFact, Linie, CodPr, Cantitate, PretUnit, Nivel, Sir) AS
        (SELECT NrFact, Linie, lf.CodPr, Cantitate, PretUnit, 0 AS Nivel,
                CAST ('\' || CAST (NrFact AS CHAR(8)) || '-' ||
                CAST (Linie AS CHAR(2)) || '-' ||  TRIM(DenPr) AS CHAR(250)) AS Sir
        FROM liniifact lf, produse p
        WHERE lf.CodPr = p.CodPr AND NrFact = 1111 AND Linie = 1
                UNION ALL
        SELECT lf.NrFact, lf.Linie, lf.CodPr, lf.Cantitate, lf.PretUnit, Nivel + 1,
                CAST ((TRIM(ierarhie.Sir) || ' \ ' || CAST (lf.NrFact AS CHAR(8)) ||
                '-' || CAST (lf.Linie AS CHAR(2)) || '-' || TRIM(p.DenPr))
                        AS CHAR(250))
        FROM ierarhie, liniifact lf, produse p
        WHERE ierarhie.NrFact = lf.NrFact AND lf.Linie = ierarhie.Linie + 1
                AND lf.CodPr = p.CodPr
)
SELECT * FROM ierarhie
```

Pe baza acestor interogări, ne putem gândi la o soluţie care să ne afişeze toate produsele vândute în fiecare factură emisă în luna septembrie 2007 (vezi figura 12.35):

```
SELECT lf.*, SYS_CONNECT_BY_PATH(lf.NrFact || '-' || lf.Linie || '-' || DenPr, ' \ ')
        AS Liniarizare, LEVEL
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
        EXTRACT (MONTH FROM DataFact) = 9 AND
        Linie = (SELECT MAX(Linie) FROM liniifact WHERE NrFact = lf.NrFact)
START WITH Linie = 1
CONNECT BY PRIOR NrFact = NrFact AND PRIOR Linie = Linie-1
ORDER BY 1, 2
```

NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	LINIARIZARE	LEVEL
3111	3	5	510	7060	\ 3111-1-Produs 1 \ 3111-2-Produs 2 \ 3111-3-Produs 5	3
3112	2	3	65	750	\ 3112-1-Produs 2 \ 3112-2-Produs 3	2
3113	1	2	120	975	\ 3113-1-Produs 2	1
3115	1	2	110	925	\ 3115-1-Produs 2	1
3116	1	2	135	930	\ 3116-1-Produs 2	1
3117	2	1	110	950	\ 3117-1-Produs 2 \ 3117-2-Produs 1	2
3118	2	1	120	930	\ 3118-1-Produs 2 \ 3118-2-Produs 1	2
3119	4	5	755	6300	\ 3119-1-Produs 2 \ 3119-2-Produs 3 \ 3119-3-Produs 4 \ 3119-4-Produs 5	4

Figura 12.35. Lista produselor vândute în fiecare factură

Iată şi versiunea SQL Server:

```
WITH ierarhie (NrFact, Linie, CodPr, Cantitate, PretUnit, Nivel, DenPr) AS
    (SELECT NrFact, Linie, lf.CodPr, Cantitate, PretUnit, 0 AS Nivel,
          CAST ('\' + CAST (NrFact AS VARCHAR) + '-' +
          CAST (Linie AS VARCHAR) + '-' + DenPr AS VARCHAR(500))
    FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
    WHERE Linie = 1 AND NrFact IN (SELECT NrFact FROM facturi
          WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9)
          UNION ALL
    SELECT lf.NrFact, lf.Linie, lf.CodPr, lf.Cantitate, lf.PretUnit, Nivel + 1,
          CAST (ierarhie.DenPr + ' \ ' + + CAST (lf.NrFact AS VARCHAR) +
          '-' + CAST (lf.Linie AS VARCHAR) + '-' + p.DenPr AS VARCHAR(500))
    FROM liniifact lf INNER JOIN ierarhie ON lf.NrFact = ierarhie.NrFact AND
          lf.Linie = ierarhie.Linie + 1 INNER JOIN produse p ON lf.CodPr = p.CodPr
    WHERE lf.NrFact IN (SELECT NrFact FROM facturi
          WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9)
    )
SELECT * FROM ierarhie i1
WHERE Linie = (SELECT MAX(Linie) FROM ierarhie WHERE NrFact = i1.NrFact)
ORDER BY 1
```

Ajungând la enunţul iniţial al problemei, includem în şirul de caractere al fiecărei facturi şi cantităţile, şi preţurile unitare de vânzare ale fiecărui produs. Iată sintaxa Oracle:

```
SELECT f.NrFact, DataFact,
    SYS_CONNECT_BY_PATH(DenPr || ': ' || Cantitate || ''|| UM
          || '*' || PretUnit || 'RONi ' , '-\\-') AS Liniarizare
FROM liniifact lf
    INNER JOIN produse p ON lf.CodPr = p.CodPr
    INNER JOIN facturi f ON lf.NrFact = f.NrFact
WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
    EXTRACT (MONTH FROM DataFact) = 9 AND
    Linie = (SELECT MAX(Linie) FROM liniifact WHERE NrFact = lf.NrFact)
```

```
START WITH Linie = 1
CONNECT BY PRIOR NrFact = NrFact AND PRIOR Linie = Linie-1
ORDER BY 1, 2
```

NRFACT	DATAFACT	LINIARIZARE
3111	01-09-2007	-\\-Produs 1: 57 buc*1000RONi -\\-Produs 2: 79 kg*1050RONi -\\-Produs 5: 510 buc*7060RONi
3112	01-09-2007	-\\-Produs 2: 65 kg*1030RONi -\\-Produs 3: 65 kg*750RONi
3113	02-09-2007	-\\-Produs 2: 120 kg*975RONi
3115	02-09-2007	-\\-Produs 2: 110 kg*925RONi
3116	10-09-2007	-\\-Produs 2: 135 kg*930RONi
3117	10-09-2007	-\\-Produs 2: 150 kg*1000RONi -\\-Produs 1: 110 buc*950RONi
3118	17-09-2007	-\\-Produs 2: 39 kg*1100RONi -\\-Produs 1: 120 buc*930RONi
3119	07-09-2007	-\\-Produs 2: 35 kg*1090RONi -\\-Produs 3: 40 kg*700RONi -\\-Produs 4: 55 l*1410RONi -\\-Produs 5: 755 buc*6300RONi

Figura 12.36. Lista produselor vândute (plus cantităţile şi preţurile unitare) în fiecare factură

cu echivalentele sale în SQL Server:

```
WITH ierarhie (NrFact, Linie, CodPr, Cantitate, PretUnit, Nivel, DenPr) AS
        (SELECT NrFact, Linie, lf.CodPr, Cantitate, PretUnit, 0 AS Nivel,
                CAST ('-\\-' + DenPr + ': ' + CAST(Cantitate AS VARCHAR) + ' ' + UM
                + '*' + CAST (PretUnit AS VARCHAR) + 'RONi ' AS VARCHAR(500))
        FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE Linie = 1 AND NrFact IN (SELECT NrFact FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9)
        UNION ALL
        SELECT lf.NrFact, lf.Linie, lf.CodPr, lf.Cantitate, lf.PretUnit, Nivel + 1,
                CAST (ierarhie.DenPr + ' -\\- ' + p.DenPr + ': ' +
                CAST(lf.Cantitate AS VARCHAR) + ' ' + UM
                + '*' + CAST (lf.PretUnit AS VARCHAR) + 'RONi ' AS VARCHAR(500))
        FROM liniifact lf INNER JOIN ierarhie ON lf.NrFact = ierarhie.NrFact AND
                lf.Linie = ierarhie.Linie + 1 INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE lf.NrFact IN (SELECT NrFact FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9)
        )
SELECT * FROM ierarhie i1
WHERE Linie = (SELECT MAX(Linie) FROM ierarhie WHERE NrFact = i1.NrFact)
ORDER BY 1
```

şi DB2:

```
WITH ierarhie ( NrFact, Linie, CodPr, Cantitate, PretUnit, Nivel, DenPr) AS
        (SELECT NrFact, Linie, lf.CodPr, Cantitate, PretUnit, 0 AS Nivel,
                CAST ('-\\-' || TRIM(DenPr) || ': ' || CHAR(Cantitate,'.') || ' ' || UM
                || '*' || CHAR(PretUnit,'.') || 'RONi ' AS CHAR(240)          )
        FROM liniifact lf, produse p
        WHERE lf.CodPr = p.CodPr AND Linie = 1 AND NrFact IN
```

```
            (SELECT NrFact FROM facturi WHERE YEAR(DataFact) = 2007
                    AND MONTH(DataFact) = 9)
            UNION ALL
        SELECT lf.NrFact, lf.Linie, lf.CodPr, lf.Cantitate, lf.PretUnit, Nivel + 1,
            CAST (TRIM(ierarhie.DenPr) || ' -\\- ' || p.DenPr || ': '
            || CHAR(lf.Cantitate, '.') || ' ' || UM
            || '*' || CHAR (lf.PretUnit,'.') || 'RONi ' AS CHAR(240))
        FROM liniifact lf, ierarhie, produse p
        WHERE lf.NrFact = ierarhie.NrFact AND lf.CodPr = p.CodPr
            AND lf.Linie = ierarhie.Linie + 1 AND lf.NrFact IN
                    (SELECT NrFact FROM facturi
                    WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9)
                    )
SELECT * FROM ierarhie i1
WHERE Linie = (SELECT MAX(linie) FROM ierarhie WHERE NrFact = i1.NrFact)
ORDER BY 1
```

Răspunsul este relativ mulţumitor (chiar dacă un pic cam stufos). În „construirea" recursivităţii, ne-am folosit (fără ruşine) de atributul Linie. Nu avem, însă, bafta aceasta de fiecare dată. Spre exemplu, pentru fiecare client dorim să afişăm, sub formă de şir, toate facturile (numărul şi data emiterii) din luna septembrie 2007. Cei pentru care nu avem nicio factură, nu vor apărea în rezultat (vezi figura 12.37).

DENCL	LISTA_FACTURI
Client 1 SRL	\ 3111 (01-09-2007) \ 3115 (02-09-2007) \ 3117 (10-09-2007) \ 3118 (17-09-2007)
Client 2 SA	\ 3113 (02-09-2007)
Client 3 SRL	\ 3119 (07-09-2007)
Client 5 SRL	\ 3112 (01-09-2007)
Client 7 SRL	\ 3116 (10-09-2007)

Figura 12.37. Lista facturilor emise în septembrie 2007 pentru fiecare client

Soluţia Oracle cea mai la îndemână vine de la funcţia OLAP ROW_NUMBER:

```
WITH num_fact AS (SELECT f.*, ROW_NUMBER() OVER (
                        PARTITION BY CodCl ORDER BY NrFact) AS NrCrt
                FROM facturi f
                WHERE EXTRACT (YEAR FROM DataFact) = 2007 AND
                        EXTRACT (MONTH FROM DataFact) = 9
                    )
SELECT DenCl, MAX(SYS_CONNECT_BY_PATH(NrFact || ' (' || DataFact || ') ', ' \ '))
        AS Lista_facturi
FROM num_fact
        INNER JOIN clienti ON num_fact.CodCl = clienti.CodCl
```

```
START WITH NrCrt = 1
CONNECT BY PRIOR num_fact.CodCl = num_fact.CodCl AND PRIOR NrCrt = NrCrt-1
GROUP BY DenCl ORDER BY 1
```

Varianta sa din MS SQL Server este[5]:

```
WITH ierarhie (DenCl, Sir_Facturi, NrCrt, Nivel) AS
        (SELECT DenCl, CAST ('\ ' + CAST (NrFact AS VARCHAR) +
                ' (' + CAST (DataFact AS VARCHAR) + ')' AS VARCHAR(1000))
                        AS Sir_Facturi, NrCrt, 0 AS Nivel
        FROM
                (SELECT DenCl, NrFact, DataFact,
                        ROW_NUMBER() OVER (PARTITION BY DenCl ORDER BY
                                NrFact) AS NrCrt
                FROM clienti c INNER JOIN facturi f
                        ON c.CodCl = f.CodCl AND YEAR(DataFact) = 2007
                                AND MONTH(DataFact) = 9
                ) x
        WHERE NrCrt = 1
                        UNION ALL
        SELECT ierarhie.DenCl, CAST (ierarhie.Sir_Facturi + '\ ' +
                CAST (NrFact AS VARCHAR) + ' (' + CAST (DataFact AS VARCHAR) +
')'
                        AS VARCHAR(1000)), x.NrCrt, Nivel + 1
        FROM ierarhie INNER JOIN
                (SELECT DenCl, NrFact, DataFact,
                        ROW_NUMBER() OVER (PARTITION BY DenCl
                        ORDER BY NrFact) AS NrCrt
                FROM clienti c INNER JOIN facturi f ON c.CodCl = f.CodCl
                        AND YEAR(DataFact) = 2007 AND MONTH(DataFact) = 9
                ) x ON ierarhie.DenCl = x.DenCl AND x.NrCrt = ierarhie.NrCrt + 1
        )
  SELECT DenCl, Sir_Facturi
  FROM ierarhie
  WHERE DenCl + CAST(NrCrt AS CHAR(3)) IN
        (SELECT DenCl + CAST (MAX(NrCrt) AS CHAR(3))
        FROM ierarhie GROUP BY DenCl)
  ORDER BY 1
```

Una dintre cele mai insolite aplicări ale interogărilor ierarhice este generarea de valori consecutive pe un interval[6], după modelul exemplului din paragraful 9.4.5!

5. Varianta DB2 vă rămâne ca temă pentru acasă.
6. Ideea am găsit-o în [Molinaro 2007], p. 322.

În Oracle, dacă dorim să creăm o tabelă „instant", în care o linie ar corespunde unei zile din luna septembrie (lucru realizat în PostgreSQL prin funcţia GENERATE_SERIES încă din finalul paragrafului 9.4), o interogare grozav de interesantă este:

```
SELECT DATE'2007-09-01' + Nr AS Zi
FROM (SELECT LEVEL AS Nr FROM dual CONNECT BY LEVEL < 30 ) Serie
```

Echivalenta interogării de mai sus în MS SQL Server este:

```
WITH serie (Nr) AS (SELECT 0 UNION ALL
                    SELECT Nr + 1 FROM serie WHERE Nr + 1 <= 29 )
SELECT CAST ('2007-09-01' AS SMALLDATETIME) + Nr FROM serie
```

în timp ce varianta DB2 se prezintă astfel:

```
WITH serie (Nr) AS (SELECT 0 AS Nr FROM SYSIBM.SYSDUMMY1 UNION ALL
                    SELECT Nr + 1 FROM serie WHERE Nr + 1 <= 29 )
SELECT CAST ('2007-09-01' AS DATE) + serie.Nr DAYS FROM serie
```

Pe acest calapod putem genera toate literele mari cuprinse între A şi Z, pentru o agendă sau orice alt gen de problemă. Iată sintaxa Oracle[7]:

```
SELECT CHR(Nr)
FROM (SELECT Nr
      FROM (SELECT LEVEL AS Nr FROM dual CONNECT BY LEVEL < 100
           ) Serie
      WHERE Nr BETWEEN 65 AND 90
      ) Litere
```

şi pe cea SQL Server[8]:

```
WITH serie (Nr) AS
    (SELECT 0 UNION ALL SELECT Nr + 1 FROM serie WHERE Nr + 1 <= 100)
SELECT CHAR(Nr) FROM serie WHERE Nr BETWEEN 65 AND 90
```

Prin urmare, putem rezolva mult mai uşor probleme precum cea din paragraful 9.4.5 (vezi figura 9.40): *Care sunt vânzările pentru toate cele zece zile cuprinse în intervalul 1-10 septembrie 2007?*

Cu sprjinul opţiunilor ierarhice/recursive, adăugăm şi următoarele soluţii:

- DB2:

```
WITH zile (Zi) AS (SELECT CAST ('2007-09-01' AS DATE) AS Zi FROM sysibm.dual
                   UNION ALL
               SELECT Zi + 1 DAY FROM zile WHERE Zi + 1 DAY <= '2007-09-10')
SELECT Zi, TRUNCATE(COALESCE(SUM(facturi.ValFact),0),0) AS Nr_Facturi
FROM zile LEFT OUTER JOIN
```

7. Anthony Molinaro, de la care am preluat ideea folosirii ierarhiilor pentru generarea intervalelor, prezintă şi o noutate post-10g: *SELECT ARRAY Nr FROM dual MODEL DIMENSION BY (0 idx) MEASURES (1 ARRAY) RULES ITERATE (10) (ARRAY [iteration_number] = iteration_number + 1).*
8. Varianta DB2 necesită adăugarea, în varianta SQL Server, după *SELECT 0* a clauzei *FROM sysibm.sysdummy1.*

```
        (SELECT f.NrFact, DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
         FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
         GROUP BY f.NrFact, DataFact) facturi
                        ON Zi = DataFact
GROUP BY Zi ORDER BY Zi
```

- Oracle:

```
SELECT Zi, TRUNC(COALESCE(SUM(facturi.ValFact),0)) AS Nr_Facturi
FROM (SELECT DATE'2007-08-31' + Nr AS Zi
        FROM (SELECT LEVEL AS Nr FROM dual CONNECT BY LEVEL < 11
                ) Serie
        ) zile LEFT OUTER JOIN
        (SELECT f.NrFact, DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
         FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
         GROUP BY f.NrFact, DataFact ) facturi
                        ON Zi = TRUNC(DataFact)
GROUP BY Zi ORDER BY Zi
```

- SQL Server:

```
WITH zile ( Zi) AS (SELECT CAST ('2007-09-01' AS SMALLDATETIME) AS Zi
                                UNION ALL
                        SELECT DATEADD(DAY, 1, Zi) FROM zile
                        WHERE DATEADD(DAY, 1, Zi) <= '2007-09-10'        )
SELECT Zi, ROUND(COALESCE(SUM(facturi.ValFact),0),0) AS Nr_Facturi
FROM zile LEFT OUTER JOIN
        (SELECT f.NrFact, DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValFact
         FROM facturi f INNER JOIN liniifact lf ON lf.NrFact = f.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
         GROUP BY f.NrFact, DataFact ) facturi
                                ON Zi = DataFact
GROUP BY Zi ORDER BY Zi
```

Pentru PostgreSQL, cea mai la îndemână opţiune rămâne, pentru acest gen de probleme, tot funcţia GENERATE_SERIES.

Ne interesează şi o problemă evocată relativ frecvent în multe cărţi/articole dedicate SQL (spre exemplu, lucrările lui Joe Celko): găsirea/umplerea golurilor pe

un anumit interval, aşa cum este cea din finalul capitolului 9: *Care sunt numerele de facturi nefolosite (în tabela FACTURI)?*

În capitolul 9 generam o tabelă ad-hoc în care pe fiecare linie era un număr întreg pozitiv cuprins între 1 şi 9999 folosind produsul cartezian. Acum tabela ad-hoc care va fi joncţionată extern cu FACTURI se obţine printr-o subconsultare ierarhică/recursivă. Iată soluţia Oracle şi o parte din rezultatul acesteia (figura 12.38):

```
SELECT Nr, NrFact, DataFact, CodCl, Obs,
        CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !'
        ELSE NULL
        END AS Situatie
FROM (SELECT Nr FROM
                (SELECT LEVEL AS Nr FROM dual
                CONNECT BY LEVEL < 999999 ) Serie
        WHERE Nr BETWEEN (SELECT MIN(NrFact) FROM facturi) AND
                                (SELECT MAX(NrFact) FROM facturi)
        ) bun LEFT OUTER JOIN facturi f ON Nr = NrFact
```

	NR	NRFACT	DATAFACT	CODCL	OBS	SITUATIE
7	1117	1117	03-08-2007	1001	(null)	(null)
8	1118	1118	04-08-2007	1001	(null)	(null)
9	1119	1119	07-08-2007	1003	(null)	(null)
10	1120	1120	07-08-2007	1001	(null)	(null)
11	1121	1121	07-08-2007	1004	(null)	(null)
12	1122	1122	07-08-2007	1005	(null)	(null)
13	1123	(null)	(null)	(null)	(null)	Numar nefolosit !
14	1124	(null)	(null)	(null)	(null)	Numar nefolosit !
15	1125	(null)	(null)	(null)	(null)	Numar nefolosit !
16	1126	(null)	(null)	(null)	(null)	Numar nefolosit !
17	1127	(null)	(null)	(null)	(null)	Numar nefolosit !

Figura 12.38. Semnalizarea numerelor nefolosite pentru facturi

Şi soluţia DB2 este una „docilă":

```
WITH serie ( Nr) AS (SELECT MIN(NrFact) AS Nr FROM facturi UNION ALL
                        SELECT Nr + 1 FROM serie
                        WHERE Nr <= (SELECT MAX(NrFact) FROM facturi))
SELECT Nr, NrFact, DataFact, CodCl, Obs,
        CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !' ELSE NULL
        END AS Situatie
FROM serie LEFT OUTER JOIN facturi f ON Nr = NrFact
```

În schimb, SQL Server este cu totul necooperant. Mai întâi, refuză execuţia pe motiv că nu poate include în SELECT-ul de definire a recursivităţii unei funcţii--agregat (SELECT MAX(NrFact)...) (vezi figura 12.39). Înlocuind „manual" valoa-

rea finală a numărului de factură, serverul o întoarce ca la Ploieşti, afirmând că interogarea ierarhică se limitează la 100 de „recursivităţi" (figura 12.40).

```
WITH serie (Nr) AS
    (SELECT CAST (MIN(NrFact) AS NUMERIC(6)) AS Nr FROM facturi
    UNION ALL
    SELECT CAST (Nr+1 AS NUMERIC(6)) FROM serie WHERE Nr <=
       (SELECT MAX(NrFact) FROM facturi)

SELECT Nr, NrFact, DataFact, CodCl, Obs,
    CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !'
    ELSE NULL
    END AS Situatie
FROM serie LEFT OUTER JOIN facturi f ON Nr=NrFact
```

```
Messages
Msg 467, Level 16, State 1, Line 1
GROUP BY, HAVING, or aggregate functions are not allowed in the recursive part
of a recursive common table expression 'serie'.
```

Figura 12.39. Prima împotrivire SQL Server (pe motiv de funcţie MAX în interogarea recursivă)

Aşa că fie lucrăm cu intervale mici, fie construim o funcţie stocată (am putea încerca treaba aceasta prin capitolul 16, dacă nu uităm).

```
WITH serie (Nr) AS
    (SELECT CAST (MIN(NrFact) AS NUMERIC(6)) AS Nr FROM facturi
     UNION ALL
     SELECT CAST (Nr+1 AS NUMERIC(6)) FROM serie WHERE Nr <= 3119)
SELECT Nr, NrFact, DataFact, CodCl, Obs,
    CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !'
    ELSE NULL
    END AS Situatie
FROM serie LEFT OUTER JOIN facturi f ON Nr=NrFact
```

```
Results   Messages
Msg 530, Level 16, State 1, Line 1
The statement terminated. The maximum recursion 100 has been exhausted before
statement completion.
```

Figura 12.40. Refuzul (de teama buclei infinite) SQL Server de a genera serii mai mari de 100 de valori

Nici funcţia CONNECTBY din PostgreSQL nu poate să ne ajute prea mult, aşa că ajungem tot la mâna funcţiei-tabelă GENERATE_SERIES:

```
SELECT Nr, NrFact, DataFact, CodCl, Obs, CASE WHEN f.NrFact IS NULL
        THEN 'Numar nefolosit !' ELSE NULL END AS Situatie
FROM (SELECT Nr
        FROM GENERATE_SERIES(0, 99999) Nr
        WHERE Nr BETWEEN (SELECT MIN(NrFact) FROM facturi) AND
```

```
            (SELECT MAX(NrFact) FROM facturi)
    ) bun
            LEFT OUTER JOIN facturi f ON Nr = NrFact
```

Capitolul 13
Tabele, restricții și actualizări prin interogări SQL

Schema minimală a unei baze de date a fost prezentată în capitolul 2, iar despre crearea tabelelor, definirea atributelor și a restricțiilor, precum și operațiunile de inserare, modificare și ștergere de linii am discutat în capitolul 3. Ceea ce aduce ca bonus acest capitol ține de folosirea consultărilor și a subconsultărilor (mai mult sau mai puțin corelate) atât pentru definirea de restricții, cât și pentru actualizarea tabelelor.

13.1. Crearea tabelelor folosind interogări

Rezultatul unei interogări, adică al unei fraze SELECT, poate fi privit ca o tabelă anonimă, temporară, ad-hoc și non-actualizabilă. În aplicații, rezultatul trebuie să fie un set de înregistrări care constituie materia primă a unui raport, dar poate fi folosit și pentru a „popula" un combo-box sau o listă dintr-un formular ș.a.m.d.

Despre crearea tabelelor am discutat pe îndelete în capitolul 3. Din rațiuni didactice și de neimplementare în cele patru servere BD, nu am prezentat câteva opțiuni din SQL:2003 ale comenzii CREATE TABLE, cum ar fi:

- atribute generate de sistem (SYSTEM GENERATED), de utilizator (USER GENERATED) sau derivate (DERIVED);
- subtabele (clauza UNDER);
- crearea unei tabele pe baza definiției (plus/minus conținutul) unei alte tabele (LIKE).

După ce am deprins elementele esențiale ale interogărilor SQL, suntem în măsură să atacăm și câteva asemenea probleme. Astfel, dacă am vrea să creăm o tabelă cu numele FACTURI_2006 cu aceleași atribute ca ale tabelei FACTURI, putem folosi comanda:

```
CREATE TABLE facturi_2006 LIKE facturi
```

Sintaxa este implementată în DB2, PostgreSQL-ul reclamând plasarea clauzei LIKE între paranteze:

```
CREATE TABLE facturi_2006 (LIKE facturi)
```

Noua tabelă va „moșteni" din sursă doar atributele, nu și restricțiile; în această privință standardul SQL:2003 și DB2 permit doar preluarea valorilor implicite (și a atributelor identitate):

```
DROP TABLE facturi_2006;
CREATE TABLE facturi_2006 LIKE facturi INCLUDING DEFAULTS;
```

PostgreSQL-ul este mai generos, întrucât formatul general permite preluarea atât a valorilor implicite, cât şi a unei serii de restricţii:

```
DROP TABLE facturi_2006;
CREATE TABLE facturi_2006 (LIKE facturi INCLUDING DEFAULTS
        INCLUDING CONSTRAINTS);
```

Clauza INCLUDING CONSTRAINTS poate să inducă în eroare: în realitate, numai restricţiile de tip CHECK sunt preluate în noua tabelă.

O altă modalitate de creare a unei tabele face apel la clauza AS. Iată sintaxa DB2:

```
DROP TABLE facturi_2006;
CREATE TABLE facturi_2006 AS
        (SELECT *
        FROM facturi) DEFINITION ONLY;
```

În Oracle, pentru a crea tabela cu atributele din sursă, sintaxa trebuie să includă clauza WHERE în care predicatul are valoarea logică *fals*:

```
DROP TABLE facturi_2006;
CREATE TABLE facturi_2006 AS
        (SELECT * FROM facturi WHERE 1 = 2);
```

În SQL Server, comanda CREATE TABLE nu permite folosirea clauzei AS şi a unei consultări; în schimb, comanda SELECT prezintă clauza INTO:

```
SELECT *
INTO facturi_2006
FROM facturi WHERE 1 = 2
```

După cum îi spune şi numele, FACTURI_2006 a fost gândită ca o tabelă-arhivă în care se păstrează toate facturile (înregistrările tabelei FACTURI) emise pe parcursul anului 2006, aşa încât avem nevoie de conţinut. Întrucât am creat structura (fără restricţii), nu ne rămâne decât să adăugăm conţinutul. Paragraful 13.3.1 este special dedicat folosirii subconsultărilor în comanda INSERT, aşa că acum doar amintim comanda. Iată o sintaxă DB2/PostgreSQL/SQL Server:

```
INSERT INTO facturi_2006
        SELECT *
        FROM facturi
        WHERE DataFact BETWEEN '2006-01-01' AND '2006-12-31'
```

şi o alta Oracle/PostgreSQL:

```
INSERT INTO facturi_2006
        SELECT *
        FROM facturi
        WHERE DataFact BETWEEN DATE'2006-01-01' AND DATE'2006-12-31'
```

Comanda CREATE TABLE permite în Oracle şi PostgreSQL crearea tabelei FACTURI_2006 şi popularea sa, realizabile dintr-o singură mişcare:

```
DROP TABLE facturi_2006;
CREATE TABLE facturi_2006 AS
(SELECT * FROM facturi WHERE EXTRACT (YEAR FROM DataFact) = 2006);
```

În MS SQL Server, comanda CREATE TABLE nu permite folosirea clauzei AS şi a unei consultări; în schimb, comanda SELECT prezintă clauza INTO:

```
SELECT *
INTO facturi_2006
FROM facturi
WHERE YEAR(DataFact) = 2006
```

Vă întrebați, fireşte, cât de frecvente sunt situațiile în care trebuie să creăm tabele pe baza altor tabele, prin subconsultări? Prefer să ofer un răspuns evaziv, mai ales că pentru o parte dintre aceste cazuri (de exemplu, furnizarea unui set de înregistrări pentru un raport, substituirea variabilelor publice - inexistente în unele SGBD-uri - etc.) pot fi găsite soluții mai bune bazate pe tabele temporare şi/sau virtuale (vezi capitolul următor) sau proceduri stocate (vezi capitolul 16).

Noi începem cu o problemă suficient de realistă - arhivarea datelor de pe anii anteriori pentru tabele „tranzacționale", cum ar fi: FACTURI, LINIIFACT, ÎNCA-SĂRI şi ÎNCASFACT. Unii practicieni vor prefera ca arhivarea să fie făcută pe o structură denormalizată. De exemplu, în tabela FACTURI_2007 să includem în înregistrările tabelei FACTURI corespunzătoare anului calendaristic 2007 şi trei atribute calculate care, ulterior, ne-ar scuti de o serie de jocțiuni - *NrLinii, ValTotală* şi *TVAFact* -, iar în LINIIFACT_2007 să adăugăm, *DenPr* şi *TVALinie*. Zis şi făcut.

În DB2, soluția ar putea arăta ca un script compus din trei comezi SQL, CREATE TABLE, INSERT şi COMMIT:

```
CREATE TABLE facturi_2007 AS
        (SELECT f.*, COUNT(*) AS NrLinii,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValTotala,
                SUM(Cantitate * PretUnit * ProcTVA) AS TVAFact
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE YEAR(DataFact) = 2007
        GROUP BY f.NrFact, DataFact, CodCl, Obs
        ) DEFINITION ONLY;

CREATE TABLE liniifact_2007 AS
        (SELECT lf.*, DenPr, UM, Cantitate * PretUnit * ProcTVA AS TVALinie
        FROM liniifact lf
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE NrFact IN
                (SELECT NrFact FROM facturi
                WHERE YEAR(DataFact) = 2007)
        ) DEFINITION ONLY;

INSERT INTO facturi_2007
```

```
        (SELECT f.*, COUNT(*) AS NrLinii,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValTotala,
                SUM(Cantitate * PretUnit * ProcTVA) AS TVAFact
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE YEAR(DataFact) = 2007
        GROUP BY f.NrFact, DataFact, CodCl, Obs
        );

INSERT INTO liniifact_2007
        (SELECT lf.*, DenPr, UM, Cantitate * PretUnit * ProcTVA AS TVALinie
        FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE NrFact IN
                (SELECT NrFact FROM facturi
                WHERE YEAR(DataFact) = 2007)
        );
COMMIT;
```

Soluţia Oracle este comună cu cea PostgreSQL, creând şi populând tabela dintr-o mişcare:

```
CREATE TABLE facturi_2007 AS
        (SELECT f.*, COUNT(*) AS NrLinii,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValTotala,
                SUM(Cantitate * PretUnit * ProcTVA) AS TVAFact
        FROM facturi f
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE EXTRACT (YEAR FROM DataFact) = 2007
        GROUP BY f.NrFact, DataFact, CodCl, Obs
        );
CREATE TABLE liniifact_2007 AS
        (SELECT lf.*, DenPr, UM, Cantitate * PretUnit * ProcTVA AS TVALinie
        FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE NrFact IN
                (SELECT NrFact FROM facturi
                WHERE EXTRACT (YEAR FROM DataFact) = 2007
                )
        )
```

Soluţia echivalentă funcţionabilă în SQL Server se prezintă astfel:

```
SELECT f.*, COUNT(*) AS NrLinii,
```

```
        SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS ValTotala,
        SUM(Cantitate * PretUnit * ProcTVA) AS TVAFact
INTO facturi_2007
FROM facturi f
        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE YEAR(DataFact) = 2007
GROUP BY f.NrFact, DataFact, CodCl, Obs;

SELECT lf.*, DenPr, UM, Cantitate * PretUnit * ProcTVA AS TVALinie
INTO liniifact_2007
FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
WHERE NrFact IN
        (SELECT NrFact
         FROM facturi
         WHERE YEAR(DataFact) = 2007
        );
```

Pentru a face lucrurile un pic mai interesante, în problema traseelor introducem un orar al curselor. Modificăm tabela DISTANȚE introducând un atribut Durata de tip INTERVAL DAY TO SECOND (în Oracle și PostgreSQL, singurele care acceptă tipuri INTERVAL), care va păstra minutele și secundele necesare parcurgerii distanțelor dintre două localități. Această durată depinde nu numai de distanța efectivă, ci și de limitele de viteză și, nu în ultimul rând, de numărul de gropi (număr rotunjit la ordinul sutelor). Apoi ștergem și repopulăm tabele (vezi listingul 13.1).

Listingul 13.1. Modificarea structurii și repopularea tabelei DISTANȚE

```
ALTER TABLE distante ADD (Durata INTERVAL DAY TO SECOND)[1];
DELETE FROM distante;
INSERT INTO distante VALUES ('Iasi', 'Vaslui', 71, INTERVAL '1:25' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Iasi', 'Tg.Frumos', 53, INTERVAL '1:05' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Tg.Frumos', 'Roman', 40, INTERVAL '0:50' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Roman', 'Bacau', 41, INTERVAL '0:47' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Roman', 'Vaslui', 80, INTERVAL '1:50' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Vaslui', 'Barlad', 54, INTERVAL '0:53' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Barlad', 'Tecuci', 48, INTERVAL '0:47' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Tecuci', 'Tisita', 20, INTERVAL '0:18' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Tisita', 'Focsani', 13, INTERVAL '0:15' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Bacau', 'Adjud', 60, INTERVAL '0:50' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Bacau', 'Vaslui', 85, INTERVAL '2:05' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Adjud', 'Tisita', 32, INTERVAL '0:40' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Focsani', 'Galati', 80, INTERVAL '1:50' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Focsani', 'Braila', 86, INTERVAL '1:57' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Focsani', 'Rm.Sarat', 42, INTERVAL '0:50' HOUR TO MINUTE);
```

1. În DB2 și MS SQL Server, comanda ALTER TABLE va avea forma: *ALTER TABLE distante ADD Durata NUMERIC(5)*, iar în comenzile INSERT durata va fi exprimată printr-un număr întreg ce reprezintă minutele; sintaxa din PostgreSQL este: *ALTER TABLE distante ADD Durata INTERVAL DAY TO SECOND* (INSERT-urile sunt identice celor din Oracle/PostgreSQL).

```
INSERT INTO distante VALUES ('Rm.Sarat', 'Braila', 84, INTERVAL '1:55' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Braila', 'Galati', 11, INTERVAL '0:25' HOUR TO MINUTE);
INSERT INTO distante VALUES ('Tecuci', 'Galati', 67, INTERVAL '1:09' HOUR TO MINUTE);
```

Aflarea numărului de minute necesare pentru a ajunge la fiecare dintre localităţile (staţiile) de pe ruta Iaşi-Bârlad-Tecuci-Focşani ar presupune în Oracle execuţia unei interogări de genul:

```
WITH tab AS (
        SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel, d.*, ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord)
SELECT DISTINCT t2.traseu, t2.Nivel AS LocNr, t2.Loc2,
        (SELECT SUM(Durata) FROM tab WHERE t2.Traseu LIKE Traseu || '%' ) AS
Km
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '**Iasi**Vaslui%Focsani'
AND t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2
```

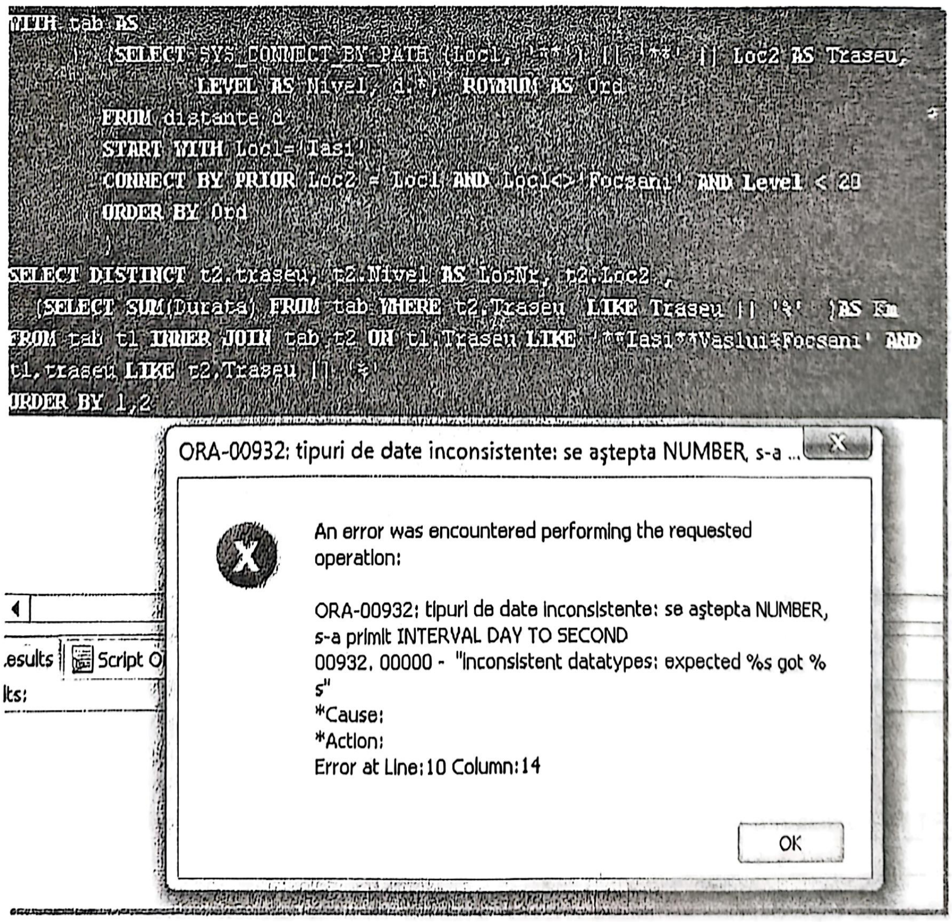


Figura 13.1. Funcţia SUM nu poate fi aplicată (în Oracle) unui atribut de tip INTERVAL

Din păcate, Oracle nu știe să adune intervale cu funcția SUM din dotare - vezi mesajul din figura 13.1, care ne spune că SUM-ul se aștepta la un argument numeric, nu la surpriza-surpriza noastră. De aceea suntem nevoiți să improvizăm un pic, convertind distanța într-un număr întreg de minute, după celebra formulă *NrTotalMinute = NrOre * 60 + NrMinute*:

```
WITH tab AS
        (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel, d.*, ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord )
SELECT DISTINCT t2.traseu, t2.Nivel AS LocNr, t2.Loc2 ,
            (SELECT SUM( EXTRACT (HOUR FROM Durata) * 60 +
                    EXTRACT (MINUTE FROM durata))
            FROM tab WHERE t2.Traseu LIKE Traseu || '%' ) AS Minute
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '**Iasi**Vaslui%Focsani'
        AND t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2
```

TRASEU	LOCNR	LOC2	MINUTE
IasiVaslui	1	Vaslui	85
IasiVaslui**Birlad	2	Birlad	138
IasiVaslui**Birlad**Tecuci	3	Tecuci	185
IasiVaslui**Birlad**Tecuci**Tisita	4	Tisita	203
IasiVaslui**Birlad**Tecuci**Tisita**Focsani	5	Focsani	218

Figura 13.2. Numărul de minute de la plecare până la fiecare stație de pe ruta Iași-Bârlad-Focșani

Rezultatul (vezi figura 13.2) ne pregătește pentru ceea ce urmează - crearea unei tabele în care, pentru început, vom stoca orarul unei curse Iași-Focșani care pleacă la ora 9:05 din Iași, pe data de 1 aprilie 2008. Întrucât clauza INTERVAL poate folosi doar literali, iar noi trebuie să ținem cont de valorile nou-introdusului atribut Durata, pentru a calcula orele de sosire/plecare în fiecare stație suntem nevoiți să recurgem la funcția TO_DSINTERVAL:

```
WITH tab AS
        (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel, d.*, ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord
        )
```

```
SELECT 1 AS "StatieNr", 'Iasi' AS "Statie",
        TIMESTAMP'2008-04-01 09:05:00' AS "Data/Ora" FROM dual
UNION
SELECT t2.Nivel + 1, t2.Loc2, TIMESTAMP'2008-04-01 09:05:00' +
        TO_DSINTERVAL ('0 ' || FLOOR(
        (SELECT SUM( EXTRACT (HOUR FROM Durata) * 60 + EXTRACT (MINUTE
                FROM durata)) FROM tab WHERE t2.Traseu LIKE Traseu || '%')/60)
                || ':' ||
        MOD( (SELECT SUM( EXTRACT (HOUR FROM Durata) * 60 +
                EXTRACT (MINUTE FROM durata)) FROM tab WHERE t2.Traseu
                        LIKE Traseu || '%' ), 60)
                || ':00')
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '**Iasi**Vaslui%Focsani'
        AND t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2
```

Din păcate, folosind CREATE TABLE AS (SELECT...) nu mai putem recurge la expresia-tabelă, ceea ce complică sintaxa comenzii. Vom începe, totuşi, prin a crea o tabelă-nomenclator a curselor în care introducem o linie:

```
CREATE TABLE curse AS (
        SELECT 10001 AS IdCursa, Traseu,
                TIMESTAMP'2008-04-01 09:05:00' AS DataOra_Plecare
        FROM
                (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' ||
                        Loc2 AS Traseu, LEVEL AS Nivel, d.*, ROWNUM AS Ord
                FROM distante d
                START WITH Loc1 = 'Iasi'
                CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani'
                        AND Level < 20
                ORDER BY Ord) tab
        WHERE Traseu LIKE '**Iasi**Vaslui%Focsani'
        );
```

Pentru a nu fi nevoiţi să mă credeţi (doar) pe cuvânt, figura 13.3 conţine linia cu pricina. În continuare, trecem la miezul problemei, propunându-ne să creăm tabela CURSE_STAŢII în care vor fi introduse toate localităţile şi orele corespunzătoare de oprire ale cursei 10001. Faptul că dispunem de tabela CURSE ne simplifică, în oarecare măsură, interogarea:

```
CREATE TABLE curse_statii AS (
        SELECT IdCursa, 1 AS "StatieNr", 'Iasi' AS Statie,
                DataOra_Plecare AS Data_Ora
        FROM curse
```

```
        WHERE IdCursa = 10001
UNION
SELECT IdCursa, Nivel + 1, Loc2, DataOra_Plecare + TO_DSINTERVAL ('0 ' ||
        FLOOR( (SELECT SUM( EXTRACT (HOUR FROM Durata) * 60 +
                        EXTRACT (MINUTE FROM durata)) FROM
                        (
                          SELECT  SYS_CONNECT_BY_PATH  (Loc1,  '**')  ||  '**'  ||
                          Loc2
                                AS Traseu, LEVEL AS Nivel, d.*,  ROWNUM AS Ord
                          FROM distante d
                                START WITH Loc1 = 'Iasi'
                                CONNECT BY PRIOR Loc2 = Loc1
                                        AND Loc1<>'Focsani' AND Level < 20
                                ORDER BY Ord
                        ) tab1
        WHERE tab.Traseu  LIKE tab1.Traseu || '%' ) / 60) || ':' ||
        MOD ( (SELECT SUM( EXTRACT (HOUR FROM Durata) * 60 +
                EXTRACT (MINUTE FROM durata)) FROM
                (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2
                        AS Traseu, LEVEL AS Nivel, d.*,  ROWNUM AS Ord
                FROM distante d
                START WITH Loc1 = 'Iasi'
                CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND
                        Level < 20
                ORDER BY Ord
                ) tab2
        WHERE tab.Traseu  LIKE tab2.Traseu || '%' ), 60) || ':00')
FROM
        (SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
                LEVEL AS Nivel, d.*,  ROWNUM AS Ord
        FROM distante d
        START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord
        ) tab INNER JOIN curse ON curse.Traseu LIKE tab.Traseu || '%')
```

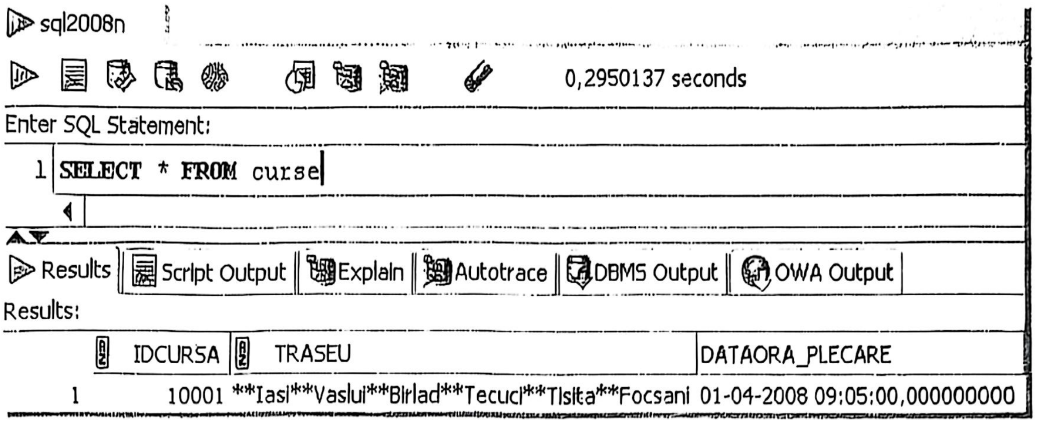


Figura 13.3. Prima (şi singura) linie din proaspăt creata tabelă CURSE

Figura 13.4 ne convinge că efortul fizic şi intelectual depus pentru redactarea acestei comenzi a meritat.

IDCURSA	StatieNr	STATIE	DATA_ORA
10001	1	Iasi	01-04-2008 09:05:00
10001	2	Vaslui	01-04-2008 10:30:00
10001	3	Birlad	01-04-2008 11:23:00
10001	4	Tecuci	01-04-2008 12:10:00
10001	5	Tisita	01-04-2008 12:28:00
10001	6	Focsani	01-04-2008 12:43:00

Figura 13.4. Staţiile şi ora de sosire/plecare pentru cursa Iaşi-Focşani din 1 aprilie 2008

În SQL Server, rezultatul din figura 13.2 se obţine prin interogarea:

```
WITH ierarhie (Cale, Nivel, Durata, Loc2) AS (
        SELECT CAST(Loc1 + ' -> ' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
                        CAST (Durata AS NUMERIC), Loc2
        FROM distante
        WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + ' -> ' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Durata + d.Durata AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20)
SELECT i2.Cale, i2.Nivel + 1 AS LocNr, i2.Loc2, i2.Durata AS Minute
FROM ierarhie i1 INNER JOIN ierarhie i2 ON i1.Cale LIKE 'Iasi -> Vaslui%Focsani' AND
                i1.Cale LIKE i2.Cale + '%'
ORDER BY 1,2
```

Cele două tabele, CURSE şi CURSE_STAŢII se obţin printr-o interogare ceva mai puţin laborioasă decât cea din Oracle:

```
WITH ierarhie (Cale, Nivel, Durata, Loc2) AS (
        SELECT CAST(Loc1 + '**' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
```

```
        CAST (Durata AS NUMERIC), Loc2 FROM distante WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + '**' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Durata + d.Durata AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20)
SELECT 1001 AS IdCursa, Cale AS Traseu,
        CAST ('2008-04-01 09:05:00' AS DATETIME) AS DataOra_Plecare
INTO curse
FROM ierarhie
WHERE Cale LIKE 'Iasi**Vaslui%Focsani';

WITH ierarhie (Cale, Nivel, Durata, Loc2) AS        (
        SELECT CAST(Loc1 + '**' + Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Durata AS NUMERIC), Loc2
        FROM distante WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.Cale + '**' + d.Loc2 AS VARCHAR(500)),
                Nivel + 1, CAST (i.Durata + d.Durata AS NUMERIC), d.Loc2
        FROM ierarhie i INNER JOIN distante d ON i.Loc2 = d.Loc1
        WHERE Nivel < 20)
SELECT IdCursa, 1 AS StatieNr, 'Iasi' AS Statie, DataOra_Plecare AS DataOra
INTO curse_statii
FROM curse WHERE IdCursa = 1001
UNION
SELECT IdCursa, Nivel + 1 AS StatieNr, Loc2,
        DATEADD(MINUTE, Durata, DataOra_Plecare)
FROM ierarhie INNER JOIN curse ON curse.Traseu LIKE ierarhie.Cale + '%'
ORDER BY 1,2
```

Beneficiind de experiența DB2 în materie de interogări ierarhice dobândită în paragraful 12.4, suntem în stare să formulăm interogarea pentru obținerea rezultatului din figura 13.2:

```
WITH ierarhie (Cale, Nivel, Durata, Loc2) AS(
        SELECT CAST(Loc1 || ' -> ' || Loc2 AS VARCHAR(500)), 0 AS Nivel,
                CAST (Durata AS NUMERIC), Loc2
        FROM distante
        WHERE Loc1 = 'Iasi'
                UNION ALL
        SELECT CAST (i.cale || ' -> ' || d.Loc2 AS VARCHAR(500)),
```

```
            Nivel + 1, CAST (i.durata + d.durata AS NUMERIC), d.loc2
        FROM ierarhie i, distante d
        WHERE i.Loc2 = d.Loc1 AND Nivel < 20)
SELECT i2.cale, i2.nivel + 1 AS LocNr, i2.Loc2, i2.Durata AS Minute
FROM ierarhie i1 INNER JOIN ierarhie i2
        ON i1.Cale LIKE 'Iasi -> Vaslui%Focsani' AND
            SUBSTR(i1.Cale,1,LENGTH(i2.Cale)) = i2.Cale
ORDER BY 1,2
```

Ne împotmolim, însă, atunci când dorim crearea celor două tabele - CURSE şi CURSE_STAŢII - ce am vrea să coincidă cu popularea. Deocamdată nu putem decât defini structura celor două tabele, urmând să încercăm popularea în paragraful 13.3.1.

```
CREATE TABLE curse AS           (
        SELECT 1001 AS IdCursa, CAST (' ' AS VARCHAR(200)) AS Traseu,
            CAST ('2008-04-01 09:05:00' AS TIMESTAMP) AS DataOra_Plecare
        FROM sysibm.sysdummy1
                            ) DEFINITION ONLY;
CREATE TABLE curse_statii AS (
        SELECT 1001 AS IdCursa, 1 AS StatieNr, CAST (' ' AS VARCHAR(30)) AS Statie,
            CAST ('2008-04-01 09:05:00' AS TIMESTAMP) AS DataOra
        FROM sysibm.sysdummy1
                            ) DEFINITION ONLY;
```

În PostgreSQL, funcţia CONNECTBY() are sintaxa ceva mai rigidă, aşa că, pentru a ne mai uşura munca, creăm o tabelă RUTE ce conţine traseele posibile dintre Iaşi şi orice altă localitate:

```
CREATE TABLE rute AS
        SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, Distanta,
            Durata, ierarhie.Cale, LEVEL AS Nivel
        FROM (SELECT *
            FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, '**')
             AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
            ) ierarhie, distante d
        WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2;
```

Pe baza acestei tabele, simplificăm crearea următoarelor două tabele, CURSE şi CURSE_STAŢII.

```
CREATE TABLE curse AS (
        SELECT 10001 AS IdCursa, Cale AS Traseu, TIMESTAMP'2008-04-01 09:05:00'
                AS DataOra_Plecare
        FROM rute
        WHERE Cale LIKE 'Iasi**Vaslui%Focsani'
        );
```

Înaintea trecerii la crearea tabelei CURSE_STAȚII trebuie să vă spun că gestiunea valorilor de tip INTERVAL este, în PostgreSQL, superioară Oracle! Putem să însumăm fără nicio grijă duratele:

```
CREATE TABLE curse_statii AS (
        SELECT IdCursa, 1 AS "StatieNr", 'Iasi' AS Statie,
                DataOra_Plecare AS Data_Ora
        FROM curse
        WHERE IdCursa = 10001
                UNION
        SELECT IdCursa, Nivel + 1, Loc2, DataOra_Plecare + (
                SELECT SUM(Durata)
                FROM rute r2
                WHERE curse.Traseu LIKE r2.Cale || '%' AND Nivel <= rute.Nivel)
        FROM curse INNER JOIN rute ON curse.Traseu LIKE rute.Cale || '%'
        WHERE IdCursa = 10001
        ORDER BY 2
        );
```

13.2. Restricții și aserțiuni pe bază de interogări

Posibilitatea de a crea restricții la nivel de înregistrare folosind fraze SELECT, care să returneze valoarea logică *adevărat* (TRUE), atunci când regula este respectată, și *fals* (FALSE) când regula este încălcată, a apărut în SQL din versiunea SQL-92. Joe Celko a cheltuit cu acest subiect destule pagini în mai toate cărțile sale dedicate SQL[2], iar Melton și Simon pomenesc și ei această facilitate ([Melton & Simon 2002], pp. 370-375). După cum lăsam să se înțeleagă din capitolele 2 și 3, restricțiile trebuie gândite, pe cât posibil, în faza de proiectare a aplicației, și nu după ce avem date reale preluate. Aceasta deoarece adăugarea ulterioară a unor restricții poate să intre în contradicție cu informațiile existente în bază la momentul declarării restricției. Noi însă vom trece sub tăcere acest neajuns.

De exemplu, instituim regula că *nu facem nicio vânzare vreunui client dacă nu avem în baza de date măcar o persoană de contact pentru acel client*. Vânzare înseamnă facturare (FACTURI), iar persoanele de contact sunt în tabela PERSCLIENȚI. Regula poate fi declarată pentru atributul CodCl din FACTURI după cum urmează:

```
ALTER TABLE facturi ADD CONSTRAINT ck_facturi_codcl CHECK (
        1 <= ALL (
                SELECT COUNT(persclienti.CodCl)
                FROM
                        (SELECT CodCl
```

2. În primul rând în [Celko 2000].

```
                FROM facturi
                GROUP BY CodCl) clienti_fact
                LEFT OUTER JOIN
                        persclienti ON clienti_fact.CodCl = persclienti.CodCl
        GROUP BY clienti_fact.CodCl
        )
                                                    )
```

Ar fi fost minunat să existe un mecanism prin care să luăm în calcul numai clientul din factura curentă (inserată sau modificată), numai că standardul (sau, cel puţin, Celko, Melton & Simon şi alţii) nu face trimitere la o asemenea posibilitate, întrucât SQL-ul este un limbaj orientat pe seturi, noţiunea de *linie curentă* fiind în disonanţă cu preceptele modelului relaţional.

Continuăm cu un exemplu care pare mai ciudat: *în tabela PERSONAL2 trebuie să existe musai nici mai mult, nici mai puţin de un director general.* Directorul general este cel care are NULL ca valoare a atributului MarcăŞef. Iată care ar putea fi restricţia:

```
ALTER TABLE personal2 ADD CONSTRAINT il_capo_di_tutti_capi CHECK (
        1 = ALL (SELECT COUNT(*) FROM personal2 WHERE MarcaSef IS NULL)
```

Ne întrebăm ce se întâmplă dacă, urmând sfatul înţelepţilor, declarăm această restricţie la crearea tabelei PERSONAL2 când, fatalitate!, nu există nicio linie (deci, nici un şef) în tabelă. Ei bine, nu-i bai! Restricţia de mai sus trebuie respectată de conţinutul tabelei, conţinut care, la momentul creării tabelei, nu există. Pentru a nu avea, totuşi, necazuri, prima linie care va fi inserată în PERSONAL2 va fi cea a lui *el lider maximo.*

Uneori declararea unei reguli sub forma restricţiei de tip CHECK este destul de anevoioasă, motiv pentru care standardul SQL a introdus, însă din versiunea SQL-92, noţiunea de *aserţiune.* La modul simplist, o aserţiune este o regulă definită în baza de date, dar care nu este legată de o tabelă anume. De exemplu, firma pentru care implementăm baza de date destinată vânzărilor, instituie regula ca *niciunui client să nu îi trimitem mai mult de 10 facturi pe an, iar dacă soldul vreunui client* (diferenţa dintre valoarea facturilor destinate acelui client şi valoarea încasărilor de la acel client) *este mai mare de, să zicem, 500 000 RON, atunci toate vânzările (facturile) se blochează.* Regula este cam dură, ca să nu spunem aiurită, dar, ca IT-işti, suntem doar servi (dacă eram mai răsăriţi, ne „făceam" *manageri*!). Iată care ar fi aserţiunea ce ar implementa această regulă:

```
CREATE ASSERTION blocaj_facturare CHECK ( 500000 <= ALL (
        SELECT facturat - COALESCE(Incasat,0)
        FROM (
                SELECT CodCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS
        Facturat
                FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        INNER JOIN produse p ON lf.CodPr = p.CodPr
                GROUP BY CodCl
                ) fact LEFT OUTER JOIN
```

```
        (SELECT CodCl, SUM(Transa) AS Incasat
        FROM facturi f
                INNER JOIN incasfact i ON f.NrFact = i.NrFact
        GROUP BY CodCl) incas   ON fact.CodCl = incas.CodCl
        )
                                                          )
```

Chiar dacă problemele de implementare nu sunt o sursă de anxietate pentru cei care redactează specificațiile standardelor SQL, vă imaginați că pentru respectarea, în orice moment, a unei reguli precum aceasta, consumul de resurse va fi unul destul de mare, mai ales după câțiva ani, timp în care s-au acumulat câteva milioane de înregistrări în LINIIFACT.

Unele restricții ridică temeri de alt gen. Să presupunem, de exemplu, *că în tabela SPORURI numărul de linii din orice lună calendaristică trebuie să corespundă exact numărului de angajați (din tabela PERSONAL2) ai firmei în luna respectivă.* Este un gen de problemă la care structura celor două tabele nu ne ajută prea mult, întrucât nu dispunem de atribute care să indice data angajării și data ieșirii din firmă ale fiecărui angajat. Unde mai punem că sunt persoane care se reîntorc în firmă după vreun „stagiu" în Italia, Spania sau orice țară aflată mai la vest. Putem însă reformula problema astfel: *numărul de înregistrări din tabela SPORURI pentru cea mai recentă lună calendaristică trebuie să fie egal cu numărul angajaților din tabela PERSONAL2*. Comanda de declarare a aserțiunii ar fi:

```
CREATE ASSERTION verificare_nr_angajati CHECK        (
        (SELECT COUNT(*)
         FROM sporuri
         WHERE (An || Luna) = (SELECT MAX(An || Luna) FROM sporuri)
        ) =
                (SELECT COUNT(*) FROM personal2)
                                                     )
```

Chiar și așa, știm că liniile se introduc pe rând în tabela SPORURI, deci aserțiunea pare condamnată din start. Ei bine, ca orice tip de restricție, și aserțiunea poate fi amânată (DEFERRED) până la finalul unei tranzacții. Prin urmare, dacă inserările s-ar face într-o singură tranzacție, am fi „asigurați".

Din păcate, deși generoase în intenții, nici regulile de validare bazate pe consultări, nici aserțiunile nu sunt implementare în vreunul dintre cele patru produse discutate în această carte. Acum vă întrebați de ce am consumat timpul cu ele. Ei bine, nu numai ca să mai treacă timpul, ci și pentru a discuta modalitățile de implementare ale acestui gen de reguli în capitolul dedicat declanșatoarelor (capitolul 17). Vom putea atunci (deși nu o s-o facem) implementa restricții dintre cele mai variate, cum ar fi:

- nu putem vinde mai mult de 15 bucăți din produsul 2 într-un an;
- o factură nu poate avea mai mult de zece linii;
- într-o gardă trebuie să fie, întotdeauna, un singur medic;

- cursă nu poate ajunge într-o staţie dacă nu a trecut prin staţia precedentă (de pe traseul respectiv.

PostgreSQL pune la dispoziţie un mecanism destul de interesant prin care pot fi definite reguli (RULES) pentru tabele şi tabele virtuale. Vom discuta acest mecanism în capitolul următor (îl vom aplica asupra unor tabele virtuale).

13.3. Actualizarea tabelelor prin subconsultări

În materie de actualizări (INSERT-uri, UPDATE-uri, DELETE-uri) pe bază de interogări, SQL-ul ne răsfaţă, chiar şi în condiţiile în care tabelele pe care le-am folosit până acum în această carte nu au o structura din cale-afară de spectaculoasă, iar exemplele nu pot fi prea impresionante.

13.3.1. INSERT-uri ce folosesc comenzi SELECT

Pentru a verifica „comportamentul" interogărilor SQL, inclusiv resursele consumate, este recomandabil să populăm tabelele bazei cu cât mai multe înregistrări. Aici apare un obstacol insurmontabil - lenea - deghizat onorabil în lipsă de timp. În cartea dedicată Oracle 9i ([Fotache *et al.* 2003]), am întors pe câte feţe am putut tabelele legate de salarizare (PERSONAL, PONTAJE, SPORURI şi REŢINERI). Pe acelaşi calapod putem să inserăm în mult mai modesta tabelă SPORURI înregistrări pentru luna august 2007, câte o înregistrare „din oficiu" pentru fiecare angajat. Sintaxa următoare este comună celor patru dialecte SQL:

```
INSERT INTO sporuri (marca, an, luna)
        SELECT marca, 2007, 8
        FROM personal2
```

Complicându-ne un pic situaţia, ne propunem să inserăm pentru luna noiembrie 2007 facturile din august 2007, cu unele amendamente. Mai întâi, numărul trebuie să fie unic, aşa că vom incrementa valorile NrFact cu 4000. Apoi, vom elimina din discuţie facturile care, în noiembrie 2007, ar fi întocmite sâmbăta şi duminica. Soluţia în dialectul Oracle ar fi:

```
INSERT INTO facturi (
        SELECT 4000 + NrFact, DataFact + INTERVAL '3' MONTH, CodCl, Obs
        FROM facturi
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT(MONTH FROM DataFact) = 8 AND
                RTRIM(TO_CHAR(DataFact + INTERVAL '3' MONTH, 'DAY'))
                        NOT IN ('SATURDAY', 'SUNDAY')
                )
```

Fireşte, înregistrările de mai sus nu ne folosesc prea mult dacă nu introducem şi liniile acestor facturi. Cu acest prilej, exemplificăm folosirea unei interogări ca argument al clauzei INTO:

```
INSERT INTO (SELECT NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact)
    (SELECT 4000 + NrFact, Linie, CodPr, Cantitate, PretUnit
    FROM liniifact
    WHERE NrFact IN (SELECT NrFact-4000 FROM facturi
    WHERE NrFact > 5000)
    )
```

Folosirea unei interogări-argument în clauza INTO este utilă cu precădere atunci când, prin SELECT-ul care desemnează liniile inserate, se preiau numai valorile câtorva dintre atribute.

Pentru luna decembrie (2007), ne propunem ceva mai ambițios. Tocmai pentru a nu avea aceleași vânzări precum una dintre lunile anterioare, vom „cumula" facturile din lunile august, septembrie și octombrie. Din data „originală" a fiecărei facturi vom prelua numai ziua; astfel 4 august, 4 septembrie și 4 octombrie vor deveni, fiecare, 4 decembrie. Numărătoarea va începe cu 7001, iar numerele vor fi consecutive (vezi figura 13.5, soluție Oracle):

```
INSERT INTO facturi
        (SELECT 7000 + Poz AS "Noul NrFact",
                CASE EXTRACT (MONTH FROM DataFact)
                        WHEN 8 THEN DataFact + INTERVAL '4' MONTH
                        WHEN 9 THEN DataFact + INTERVAL '3' MONTH
                        WHEN 10 THEN DataFact + INTERVAL '2' MONTH
                END AS "Noua DataFact", CodCl, Obs
        FROM
                (SELECT NrFact, RANK() OVER
                        (ORDER BY EXTRACT(DAY FROM DataFact), NrFact) AS Poz
                FROM facturi
                WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                        EXTRACT(MONTH FROM DataFact) IN (8,9,10)
                ) coresp INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                        EXTRACT(YEAR FROM DataFact) = 2007
                        AND EXTRACT(MONTH FROM DataFact) IN (8,9,10)
        )
```

NrFact vechi	DataFact veche	Ordine	Noul NrFact	Noua DataFact
1111	01-08-2007	1	7001	01-12-2007
1112	01-08-2007	2	7002	01-12-2007
1113	01-08-2007	3	7003	01-12-2007
1114	01-08-2007	4	7004	01-12-2007
3111	01-09-2007	5	7005	01-12-2007
3112	01-09-2007	6	7006	01-12-2007
1115	02-08-2007	7	7007	02-12-2007
1116	02-08-2007	8	7008	02-12-2007
3113	02-09-2007	9	7009	02-12-2007
3115	02-09-2007	10	7010	02-12-2007
1117	03-08-2007	11	7011	03-12-2007
1118	04-08-2007	12	7012	04-12-2007
1119	07-08-2007	13	7013	07-12-2007
1120	07-08-2007	14	7014	07-12-2007
1121	07-08-2007	15	7015	07-12-2007
1122	07-08-2007	16	7016	07-12-2007
3119	07-09-2007	17	7017	07-12-2007
3116	10-09-2007	18	7018	10-12-2007

Figura 13.5. Renumerotarea facturilor din lunile august, septembrie şi octombrie 2007

Inserarea rândurilor (liniilor) fiecăreia dintre facturi se realizează astfel:

```
INSERT INTO liniifact (
        SELECT 7000 + Poz AS "Noul NrFact", Linie, CodPr, Cantitate, PretUnit
        FROM
                (SELECT NrFact, RANK() OVER (ORDER BY
                        EXTRACT(DAY FROM DataFact), NrFact) AS Poz
                FROM facturi
                WHERE EXTRACT(YEAR FROM DataFact) = 2007
                        AND EXTRACT(MONTH FROM DataFact) IN (8,9,10)
                ) coresp INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                        EXTRACT(YEAR FROM DataFact) = 2007 AND
                        EXTRACT(MONTH FROM DataFact) IN (8,9,10)
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                        )
```

Am văzut în primul paragraf al acestui capitol că DB2 este destul de necooperant în materie de creare a tabelelor prin interogări. Este motivul pentru care CREATE TABLE AS poate fi folosită numai împreună cu *DEFINITION ONLY*, prin urmare tabela poate fi creată printr-un SELECT, dar fără înregistrări. Aşa că amânăm popularea pentru acest paragraf, bănuind că am putea folosi în comanda INSERT un SELECT care să folosească clauza WITH. Ei, bine, interogările următoare punctează serios la capitolul imagine:

```
INSERT INTO curse
        WITH ierarhie (Cale, Nivel, Durata, Loc2) AS        (
```

```
            SELECT CAST(Loc1 || '**' || Loc2 AS VARCHAR(500)), 0 AS Nivel,
                    CAST (Durata AS NUMERIC), Loc2
            FROM distante WHERE Loc1 = 'Iasi'
                            UNION ALL
            SELECT CAST (i.cale || '**' || d.Loc2 AS VARCHAR(500)),
                    Nivel + 1, CAST (i.durata + d.durata AS NUMERIC), d.loc2
            FROM ierarhie i, distante d
            WHERE i.Loc2 = d.Loc1 AND Nivel < 20)
        SELECT 1001 AS IdCursa, Cale AS Traseu,
            CAST ('2008-04-01 09:05:00' AS TIMESTAMP) AS DataOra_Plecare
        FROM ierarhie
        WHERE Cale LIKE 'Iasi**Vaslui%Focsani';

INSERT INTO curse_statii
        WITH ierarhie (Cale, Nivel, Durata, Loc2) AS          (
            SELECT CAST(Loc1 || '**' || Loc2 AS VARCHAR(500)), 0 AS Nivel,
                    CAST (Durata AS NUMERIC), Loc2
            FROM distante WHERE Loc1 = 'Iasi'
                    UNION ALL
            SELECT CAST (i.cale || '**' || d.Loc2 AS VARCHAR(500)),
                    Nivel + 1, CAST (i.durata + d.durata AS NUMERIC), d.loc2
            FROM ierarhie i, distante d
            WHERE i.Loc2 = d.Loc1 AND Nivel < 20)
        SELECT IdCursa, 1 AS StatieNr, 'Iasi' AS Statie, DataOra_Plecare AS DataOra
        FROM curse
        WHERE IdCursa = 1001
            UNION
        SELECT IdCursa, nivel + 1, Loc2,
            DataOra_Plecare + Durata MINUTE
        FROM ierarhie INNER JOIN curse
            ON SUBSTR(curse.Traseu,1,LENGTH(ierarhie.Cale)) = ierarhie.Cale
        ORDER BY 1,2
```

Dacă am reușit operațiunea aceasta laborioasă, popularea tabelelor FACTURI și LINIIFACT pentru luna noiembrie 2007 este floare la ureche:

```
-- DB2: preluarea, pentru luna noiembrie 2007, a continutului (aproximativ) facturilor
-- din august 2007
INSERT INTO facturi (NrFact, DataFact, CodCl, Obs) (
        SELECT 4000 + NrFact, DataFact + 3 MONTHS, CodCl, Obs
        FROM facturi
        WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 8
```

```
                AND DAYNAME(DataFact + 3 MONTHS) NOT IN
                        ('SATURDAY', 'SUNDAY')
                );
-- DB2: adaugarea liniilor facturilor "proaspete"
INSERT INTO (SELECT NrFact, Linie, CodPr, Cantitate, PretUnit FROM liniifact)
        (SELECT 4000 + NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact
        WHERE NrFact IN (
                SELECT NrFact-4000 FROM facturi WHERE NrFact > 5000)
        );
```

Nici cele două INSERT-uri dedicate facturilor din luna decembrie 2007 nu ridică probleme deosebite:

```
-- DB2: adunarea facturilor din august/septembrie/octombrie si generarea facturilor
-- pentru decembrie 2007
INSERT INTO facturi (NrFact, DataFact, CodCl, Obs)
        (SELECT 7000 + Poz AS "Noul NrFact",
                CASE MONTH(DataFact)
                        WHEN 8 THEN DataFact + 4 MONTHS
                        WHEN 9 THEN DataFact + 3 MONTHS
                        WHEN 10 THEN DataFact + 2 MONTHS
                END AS "Noua DataFact", CodCl, Obs
        FROM
                (SELECT NrFact, RANK() OVER
                        (ORDER BY DAY(DataFact), NrFact) AS Poz
                FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                ) coresp
                INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                        YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                );
-- DB2: liniile facturilor din decembrie 2007
INSERT INTO liniifact (
        SELECT 7000 + Poz AS "Noul NrFact", Linie, CodPr, Cantitate, PretUnit
        FROM
                (SELECT NrFact, RANK() OVER
                        (ORDER BY DAY(DataFact), NrFact) AS Poz
                FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                ) coresp
                        INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
```

```
            YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
            INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
            );
```

SQL Server nu permite specificarea după clauza INTO a unei interogări care să returneze atributele pentru care vor fi inserate valori, aşa că sintaxa pentru popularea cu date pentru noiembrie 2007 are forma:

```
– SQL Server: preluarea, pentru luna noiembrie 2007 a datelor din facturile lunii august
– 2007
INSERT INTO facturi
        SELECT 4000 + NrFact, DATEADD(MONTH, 3, DataFact), CodCl, Obs
        FROM facturi
        WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) = 8
                AND DATENAME(WEEKDAY, DATEADD(MONTH, 3, DataFact))
                        NOT IN ('SATURDAY', 'SUNDAY');
– SQL Server: adaugarea liniilor facturilor "proaspete"
INSERT INTO liniifact
        SELECT 4000 + NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact WHERE NrFact IN (
                SELECT NrFact-4000 FROM facturi WHERE NrFact > 5000)
```

Iată şi cele două INSERT-uri dedicate facturilor artificiale din luna decembrie 2007:

```
– SQL Server: adunarea facturilor din august/septembrie/octombrie si generarea
– facturilor pentru decembrie 2007
INSERT INTO facturi
        SELECT 7000 + Poz,
                CASE MONTH(DataFact)
                        WHEN 8 THEN DATEADD(MONTH, 4, DataFact)
                        WHEN 9 THEN DATEADD(MONTH, 3, DataFact)
                        WHEN 10 THEN DATEADD(MONTH, 2, DataFact)
                END AS "Noua DataFact", CodCl, Obs
        FROM
                (SELECT NrFact, RANK() OVER (ORDER BY DAY(DataFact), NrFact)
                        AS Poz
                FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                ) coresp
                        INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                        YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10);

– SQL Server: liniile facturilor din decembrie 2007
```

```
INSERT INTO liniifact
        SELECT 7000 + Poz, Linie, CodPr, Cantitate, PretUnit
        FROM
                (SELECT NrFact, RANK() OVER (ORDER BY DAY(DataFact), NrFact)
                        AS Poz
                FROM facturi
                WHERE YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                ) coresp
                        INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                           YEAR(DataFact) = 2007 AND MONTH(DataFact) IN (8,9,10)
                        INNER JOIN liniifact lf ON f.NrFact = lf.NrFact;
```

Sintaxa primelor două comenzi INSERT dedicate adăugării artificiale de înregistrări pentru luna noiembrie 2007 are în PostgreSQL două diferenţe faţă de cea Oracle. Mai întâi, intervalul se exprimă în maniera *INTERVAL '3 MONTH'*, şi nu *INTERVAL '3' MONTH*. În al doilea rând, ca şi în SQL Server, după INTO nu poate urma un SELECT:

```
– PostgreSQL: preluarea, pentru luna noiembrie 2007 a datelor din facturile
– lunii august 2007
INSERT INTO facturi (
        SELECT 4000 + NrFact, DataFact + INTERVAL '3 MONTH', CodCl, Obs
        FROM facturi
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
EXTRACT(MONTH FROM DataFact) = 8 AND
                RTRIM(TO_CHAR(DataFact + INTERVAL '3 MONTH', 'DAY')) NOT IN
                        ('SATURDAY', 'SUNDAY')
                );
– PostgreSQL:  adaugarea liniilor facturilor "proaspete" pentru noiembrie 2007
INSERT INTO liniifact
        (SELECT 4000 + NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact
        WHERE NrFact IN (SELECT NrFact-4000
                        FROM facturi
                        WHERE NrFact > 5000)
        );
```

Partea cea mai interesantă, cea cu agregarea facturilor din august, septembrie şi octombrie şi, astfel, generarea (artificială) a facturilor lunii decembrie 2007 se loveşte de un obstacol destul de neplăcut în PostgreSQL – inexistenţa funcţiilor OLAP, în cazul nostru a funcţiei RANK. Avem însă o soluţie bazată pe corelare, prezentată în paragraful 10.1 în exemplele cu afişarea numărului curent al fiecărei facturi (figurile 10.7 şi 10.8). Iat-o:

```
INSERT INTO facturi
```

```
        (SELECT 7000 + Poz AS "Noul NrFact",
                CASE EXTRACT (MONTH FROM DataFact)
                        WHEN 8 THEN DataFact + INTERVAL '4 MONTH'
                        WHEN 9 THEN DataFact + INTERVAL '3 MONTH'
                        WHEN 10 THEN DataFact + INTERVAL '2 MONTH'
                END AS "Noua DataFact", CodCl, Obs
        FROM
                (SELECT NrFact, (SELECT COUNT(*) + 1
                FROM facturi
                WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                    EXTRACT(MONTH FROM DataFact) IN (8,9,10) AND
                    (EXTRACT(DAY    FROM    DataFact)    <    EXTRACT(DAY    FROM
                f1.DataFact)
                        OR EXTRACT(DAY FROM DataFact) =
                        EXTRACT(DAY FROM f1.DataFact) AND NrFact < f1.NrFact)
                ) AS Poz
FROM facturi f1
WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
        EXTRACT(MONTH FROM DataFact) IN (8,9,10)
        ) coresp
        INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
                EXTRACT(YEAR FROM DataFact) = 2007
                        AND EXTRACT(MONTH FROM DataFact) IN (8,9,10)
        );

INSERT INTO liniifact (
        SELECT 7000 + Poz , Linie, CodPr, Cantitate, PretUnit
        FROM
(SELECT NrFact, (SELECT COUNT(*) + 1
FROM facturi
WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND EXTRACT(MONTH FROM
        DataFact) IN (8,9,10) AND (EXTRACT(DAY FROM
        DataFact) < EXTRACT(DAY FROM f1.DataFact) OR
        EXTRACT(DAY FROM DataFact) = EXTRACT(DAY FROM f1.DataFact)
        AND NrFact < f1.NrFact)
                ) AS Poz
        FROM facturi f1
        WHERE EXTRACT(YEAR FROM DataFact) = 2007 AND
                EXTRACT(MONTH FROM DataFact) IN (8,9,10)
                ) coresp
```

```
INNER JOIN facturi f ON coresp.NrFact = f.NrFact AND
        EXTRACT(YEAR FROM DataFact) = 2007
        AND EXTRACT(MONTH FROM DataFact) IN (8,9,10)
INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
        );
```

13.3.2. Inserări în tabele multiple

Comanda INSERT are, numai în Oracle, şi un format prin care, simultan, pot fi introduse linii în două sau mai multe tabele, pe baza unor condiţii aplicate rezultatului unei fraze SELECT. Revenim la o problemă introdusă în primul paragraf din acest capitol, dar pe care o complicăm exagerat. Ne interesează ca datele despre facturi (celebrul „tandem" FACTURI-LINIIFACT) să fie arhivate lunar. Chiar dacă noi pe anul 2007 avem introduse datele pe lunile august--decembrie (pentru ultimele luni am „semănat" câteva înregistrări chiar în paragraful precedent), imaginăm o soluţie aplicabilă pentru un an întreg. Pentru 2007 vom crea 12 de tabele-fragment pe baza tabelei FACTURI: FACTURI_2007_1, FACTURI_2007_2, ..., FACTURI_2007_12 şi 12 din LINIIFACT: LINII-FACT_2007_1, ... (vezi listingul 13.2).

Listingul 13.2. Crearea „clasică" a tabelelor de arhivare

```
CREATE TABLE facturi_2007_1 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 1);
CREATE TABLE facturi_2007_2 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 2);
CREATE TABLE facturi_2007_3 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 3);
CREATE TABLE facturi_2007_4 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 4);
CREATE TABLE facturi_2007_5 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 5);
CREATE TABLE facturi_2007_6 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 6);
CREATE TABLE facturi_2007_7 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 7);
CREATE TABLE facturi_2007_8 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 8);
CREATE TABLE facturi_2007_9 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 9);
CREATE TABLE facturi_2007_10 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 10);
CREATE TABLE facturi_2007_11 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 11);
CREATE TABLE facturi_2007_12 AS  (SELECT * FROM facturi WHERE EXTRACT
          (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 12);

CREATE TABLE liniifact_2007_1 AS (SELECT * FROM liniifact
          WHERE NrFact IN (SELECT NrFact FROM facturi_2007_1));
CREATE TABLE liniifact_2007_2 AS (SELECT * FROM liniifact
          WHERE NrFact IN (SELECT NrFact FROM facturi_2007_2));
CREATE TABLE liniifact_2007_3 AS (SELECT * FROM liniifact
          WHERE NrFact IN (SELECT NrFact FROM facturi_2007_3));
```

```
CREATE TABLE liniifact_2007_4 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_4));
CREATE TABLE liniifact_2007_5 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_5));
CREATE TABLE liniifact_2007_6 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_6));
CREATE TABLE liniifact_2007_7 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_7));
CREATE TABLE liniifact_2007_8 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_8));
CREATE TABLE liniifact_2007_9 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_9));
CREATE TABLE liniifact_2007_10 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_10));
CREATE TABLE liniifact_2007_11 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_10));
CREATE TABLE liniifact_2007_12 AS (SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi_2007_12));
```

Să presupunem că, la arhivare, câteva facturi au fost omise (din cauze neelucidate) și, pentru siguranță, reexecutăm arhivarea pentru întreg anul 2007. În locul clasicei variante DELETE/INSERT din listingul 13.3, putem încerca și ceva mai exotic, chiar dacă câștigul de productivitate nu este chiar entuziasmant.

Listingul 13.3. Repopularea tabelelor de arhivare

```
DELETE FROM facturi_2007_1;
DELETE FROM facturi_2007_2;
...

INSERT INTO facturi_2007_1 (SELECT * FROM facturi WHERE EXTRACT
    (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 1);
INSERT INTO facturi_2007_2 (SELECT * FROM facturi WHERE EXTRACT
    (YEAR FROM DataFact) = 2007 AND EXTRACT (MONTH FROM DataFact) = 2);
....

DELETE FROM liniifact_2007_1;
DELETE FROM liniifact_2007_2;
...

INSERT INTO liniifact_2007_1 (SELECT * FROM liniifact
    WHERE NrFact IN (SELECT NrFact FROM facturi_2007_1));
 INSERT INTO liniifact_2007_2 (SELECT * FROM liniifact
    WHERE NrFact IN (SELECT NrFact FROM facturi_2007_2));
...
```

Numai pentru arhivele tabelei FACTURI cele 12 comenzi INSERT pot fi înlocuite cu una singură având următoarea formă:

```
INSERT ALL
    WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-01' THEN INTO facturi_2007_1
    WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-02' THEN INTO facturi_2007_2
    WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-03' THEN INTO facturi_2007_3
    WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-04' THEN INTO facturi_2007_4
    WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-05' THEN INTO facturi_2007_5
```

```
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-06' THEN INTO facturi_2007_6
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-07' THEN INTO facturi_2007_7
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-08' THEN INTO facturi_2007_8
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-09' THEN INTO facturi_2007_9
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-10' THEN INTO facturi_2007_10
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-11' THEN INTO facturi_2007_11
	WHEN TO_CHAR(DataFact, 'YYYY-MM') = '2007-12' THEN INTO facturi_2007_12
SELECT * FROM facturi
```

O aplicare spectaculoasă a comenzii INSERT ALL este legată de „verticalizarea orizontalităţii". Să luăm un exemplu relativ apropriat de cazul înscrierii la masterat la Facultatea de Economie şi Administrarea Afacerilor. Există şase specializări de masterat, fiecare cu un anumit număr de locuri disponibile:

```
CREATE TABLE mastere (
	Spec VARCHAR2(4) PRIMARY KEY,
	DenSpec VARCHAR2(40) NOT NULL,
	NrLocuri NUMBER(3)
	);
```

La înscriere, candidaţii completează o fişă în care, pe lângă nume şi media anilor de studiu (*Media_AS*) şi media de la licenţă (*Media_Lic*), precizează, în ordinea preferinţei, programele pe care ar dori să le urmeze. Repartizarea se face în funcţie de ordinea mediei de admitere care se calculează prin formula *Med_AS * 0.6 + Med_Lic * 0.4*. Dacă un student nu „încape" la prima specializare - *Spec1* - pe care a solicitat-o (locurile de la specializarea de masterat respectivă s-au completat pentru că au fost studenţi cu medii mai mari care au avut prioritate), atunci va fi repartizat la *Spec2;* dacă şi aici locurile sunt ocupate, se va încerca repartizarea sa la *Spec3* ş.a.m.d. Dacă, însă, candidatul nu e interesat decât de două specializări, va completa *Spec1* şi *Spec2,* restul atributelor (*Spec3, ..., Spec6*) fiind NULL:

```
CREATE TABLE candidati (
	IdCandidat NUMBER(6) NOT NULL PRIMARY KEY,
	NumePren VARCHAR2(50) NOT NULL,
	Media_AS NUMBER(4,2) NOT NULL CONSTRAINT ck_media_as
		CHECK (media_as BETWEEN 5 AND 10),
	Media_Lic NUMBER(4,2) NOT NULL CONSTRAINT ck_media_lic
		CHECK (media_lic BETWEEN 6 AND 10),
	Spec1 VARCHAR2(4) NOT NULL REFERENCES mastere(Spec),
	Spec2 VARCHAR2(4) REFERENCES mastere(Spec),
	Spec3 VARCHAR2(4) REFERENCES mastere(Spec),
	Spec4 VARCHAR2(4) REFERENCES mastere(Spec),
	Spec5 VARCHAR2(4) REFERENCES mastere(Spec),
	Spec6 VARCHAR2(4) REFERENCES mastere(Spec),
	CONSTRAINT ck_spec CHECK (
```

```
COALESCE(spec2, ' ') = COALESCE(spec2, spec3, ' ') AND
COALESCE(spec3, ' ') = COALESCE(spec3, spec4, ' ') AND
COALESCE(spec4, ' ') = COALESCE(spec4, spec5, ' ') AND
COALESCE(spec5, ' ') = COALESCE(spec5, spec6, ' ')          )
        );
```

Probabil cel mai interesant punct al comenzii este regula de validare declarată la finalul comenzii. Cei mai răbdători dintre dumneavoastră și-au dat seama că acest CHECK urmărește ca, odată ce un candidat nu a completat una din opțiunile Spec2, ..., Spec5, niciuna dintre următoarele nu mai poate fi nenulă. Cu ocazia aceasta am ilustrat și sintaxa cu trei argumente a comenzii COALESCE. Listingul 13.4 populează cu câteva înregistrări cele două tabele.

Listingul 13.4. Popularea tabelelor MASTERE și CANDIDAȚI

```
DELETE FROM candidati;
DELETE FROM mastere;

INSERT INTO mastere VALUES ('SIA', 'Sisteme informationale pentru afaceri', 4);
INSERT INTO mastere VALUES ('FAB', 'Finante-Asigurari-Banci', 6);
INSERT INTO mastere VALUES ('CEA', 'Contabilitate, expertiza si audit', 6);
INSERT INTO mastere VALUES ('MARK', 'Marketing', 5);
INSERT INTO mastere VALUES ('EAI', 'Economie si afaceri internationale', 5);
INSERT INTO mastere VALUES ('MRU', 'Managementul resurselor umane', 4);

INSERT INTO candidati VALUES ( 1, 'Popescu I. Irina', 7.5, 9.50, 'FAB', 'MARK', 'EAI', 'MRU',
    NULL, NULL);
INSERT INTO candidati VALUES ( 2, 'Babias D. Ecaterina', 8.5, 9.20, 'SIA', 'EAI', NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES ( 3, 'Strat P. Iulian', 7.35, 9.50, 'CEA', 'EAI', 'MRU', 'SIA',
    'FAB', 'MARK');
INSERT INTO candidati VALUES ( 4, 'Georgescu M. Honda', 8.5, 9.00, 'MRU', 'MARK', 'EAI',
    'FAB', 'CEA', 'SIA');
INSERT INTO candidati VALUES ( 5, 'Munteanu A. Optimista', 9.5, 9.50, 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 6, 'Dumitriu F. Dura', 9.5, 10, 'EAI', NULL, NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES ( 7, 'Mesnita G. Plinutul', 10, 10, 'EAI', 'MARK', 'MRU', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES ( 8, 'Greavu V. Doru', 9.25, 8.50, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES ( 9, 'Baboi P. Iustina', 8.5, 8.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (10, 'Postelnicu I. Irina', 9.5, 8.25, 'MARK', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (11, 'Fotache H. Fanel', 9.75, 9.50, 'MRU', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (12, 'Moscovici J. Cristina', 9.80, 9.30, 'CEA', 'SIA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (13, 'Rusu I. Vanda', 7.80, 9.10, 'FAB', 'CEA', 'MARK', 'MRU',
    NULL, NULL);
INSERT INTO candidati VALUES (14, 'Spinu M. Algebra', 7.25, 9.00, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (15, 'Sandovici I. Irina', 7.05, 7.50, 'MARK', 'MRU', 'EAI', 'CEA',
    NULL, NULL);
INSERT INTO candidati VALUES (16, 'Plai V. Picior', 7.5, 7.90, 'EAI', 'SIA', 'CEA', 'FAB',
    NULL, NULL);
```

```
INSERT INTO candidati VALUES (17, 'Ambuscada B. Cristina', 8.25, 9.50, 'SIA', 'CEA', 'FAB', 'EAI',
    NULL, NULL);
INSERT INTO candidati VALUES (18, 'Pinda A. Axinia', 8.75, 9.00, 'SIA', 'FAB', 'CEA', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (19, 'Planton V. Grigore', 9.25, 9.50, 'FAB', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (20, 'Sergentu I. Zece',7.5, 9.50, 'FAB', 'CEA', NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (21, 'Ababei T. Marian-Vasile', 7.5, 9.50, 'CEA', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (22, 'Popistasu J. Maria', 7.5, 9.50, 'FAB', 'EAI', 'CEA', 'MRU',
    NULL, NULL);
INSERT INTO candidati VALUES (23, 'Plop R. Robert', 7.5, 9.50, 'FAB', 'MARK', 'MRU', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (24, 'Aflorei H. Crina',7.5, 9.50, 'EAI', 'SIA', NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (25, 'Afaunei P. Gina',7.5, 9.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (26, 'Vatamanu I. Alexandrina', 7.5, 9.50, 'MRU', 'MARK', 'EAI',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (27, 'Lovin P. Marian',7.5, 9.50, 'MARK', 'MRU', NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (28, 'Antiteza W. Florin', 7.5, 9.50, 'EAI', 'MARK', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (29, 'Prepelita V. Ion', 7.5, 9.50, 'EAI', NULL, NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (30, 'Cioara X. Sanda', 7.5, 9.50, 'CEA', 'FAB', 'EAI', NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (31, 'Metafora Y. Vasile',7.5, 9.50, 'SIA', 'CEA', NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (32, 'Strasina R. Elvis', 7.5, 9.50, 'CEA', NULL, NULL, NULL,
    NULL, NULL);
INSERT INTO candidati VALUES (33, 'Durere V. Vasile', 7.5, 9.50, 'FAB', 'CEA', 'EAI', 'MARK',
    'MRU', NULL);
INSERT INTO candidati VALUES (34, 'Sedentaru L. Marius-Daniel',7.5, 9.50, 'MARK', 'MRU',
    'EAI', NULL, NULL, NULL);
INSERT INTO candidati VALUES (35, 'Zgircitu I. Daniel',7.5, 9.50, 'MRU', NULL, NULL, NULL,
    NULL, NULL);
```

Pentru a putea prelucra opţiunile fiecărui candidat (vezi capitolul 16) şi a determina specializarea de masterat la care a fost admis un candidat (vor exista, însă, şi candidaţi respinşi), vom crea o tabelă specială - OPŢIUNI - cu atributele *IdCandidat, Ordine* şi *Spec.* Tot din comoditate, în locul clasicei variante CREATE TABLE ... listă atribute şi restricţii... vom apela la o interogare:

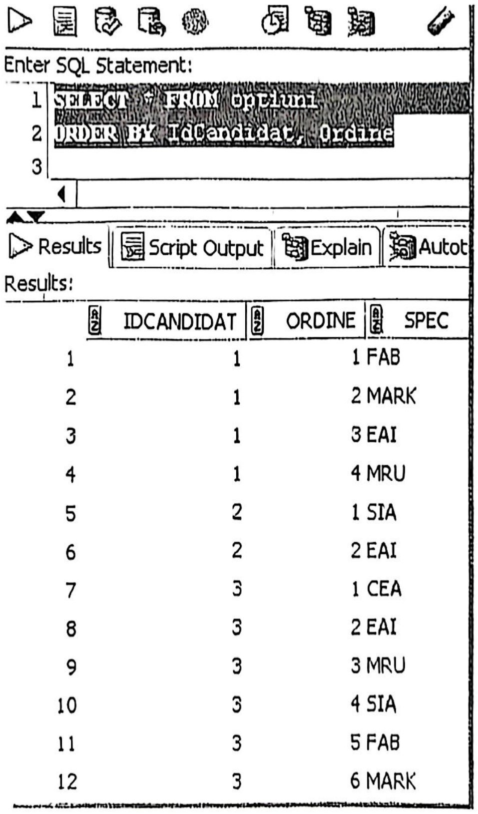
```
CREATE TABLE optiuni AS
        SELECT IdCandidat, 99 AS Ordine, Spec1 AS Spec
        FROM candidati
        WHERE 1 = 2
```

Condiţia din clauza WHERE ne va asigura că, la creare, tabela OPŢIUNI nu va avea nicio linie. Popularea cu înregistrări a acestei noi tabele poate fi realizată dintr-o suflare astfel:

```
INSERT ALL
        WHEN Spec1 IS NOT NULL THEN INTO optiuni
```

```
        VALUES (IdCandidat, 1, Spec1)
WHEN Spec2 IS NOT NULL THEN INTO optiuni
        VALUES (IdCandidat, 2, Spec2)
WHEN Spec3 IS NOT NULL THEN INTO optiuni
        VALUES (IdCandidat, 3, Spec3)
WHEN Spec4 IS NOT NULL THEN INTO optiuni
        VALUES (IdCandidat, 4, Spec4)
WHEN Spec5 IS NOT NULL THEN INTO optiuni
        VALUES (IdCandidat, 5, Spec5)
WHEN Spec6 IS NOT NULL THEN INTO optiuni
        VALUES (IdCandidat, 6, Spec6)
SELECT * FROM candidati
```

Pentru confirmare, figura 13.6 prezintă câteva dintre liniile tabelei.



	IDCANDIDAT	ORDINE	SPEC
1	1	1	FAB
2	1	2	MARK
3	1	3	EAI
4	1	4	MRU
5	2	1	SIA
6	2	2	EAI
7	3	1	CEA
8	3	2	EAI
9	3	3	MRU
10	3	4	SIA
11	3	5	FAB
12	3	6	MARK

Figura 13.6. Câteva linii din tabela OPȚIUNI

13.3.3. Ștergeri ceva mai sofisticate

După cum am văzut în capitolul 3, comenzile de modificare (UPDATE) și de ștergere (DELETE) prezintă clauzele FROM și WHERE similare frazei SELECT. După discuțiile din capitolele precedente suntem în măsură să formulăm condiții mai sofisticate de ștergere a unor linii. Astfel, dacă dorim să eliminăm din baza de

date toate facturile care nu au nicio linie, putem folosi cel puţin trei variante. Prima este şi cea mai „cuminte":

```
DELETE FROM facturi
WHERE NrFact NOT IN (SELECT NrFact FROM liniifact)
```

A doua „corelează" (un pic):

```
DELETE FROM facturi
WHERE NOT EXISTS
        (SELECT 1  FROM liniifact
        WHERE liniifact.NrFact = facturi.NrFact)
```

Mai spectaculoasă este însă varianta în care, în locul numelui unei tabele, se foloseşte o subconsultare în clauza FROM, sintaxă care este, însă, acceptată numai de DB2 şi Oracle:

```
DELETE FROM (
        SELECT *
        FROM facturi
        WHERE NrFact NOT IN (SELECT NrFact FROM liniifact)
        )
```

Interesant este că interogarea aceasta funcţionează şi dacă în locul calificatorului pentru toate atributele tabelei (*) am fi folosit NrFact sau oricare alt atribut (chiar şi Obs care prezintă valori nule):

```
DELETE FROM (SELECT NrFact FROM facturi WHERE NrFact NOT IN
        (SELECT NrFact FROM liniifact))
```

Întrucât am făcut arhivarea facturilor pentru lunile anului 2007, dorim să ştergem liniile arhivate din tabelele FACTURI şi LINIIFACT. Să luăm exemplul lunii iulie, sintaxa fiind una comună Oracle/DB2:

```
DELETE FROM (SELECT * FROM liniifact) lf
WHERE EXISTS
        (SELECT 1 FROM facturi_2007_7 f07 WHERE f07.NrFact = lf.NrFact);

DELETE FROM (SELECT * FROM facturi) f
WHERE EXISTS
        (SELECT 1 FROM facturi_2007_7 f07 WHERE f07.NrFact = f.NrFact);
```

În cartea dedicată Oracle 9i am prezentat o găselniţă interesantă - includerea în clauza FROM a unei comenzi DELETE a joncţiunii unei tabele. Astfel, comanda:

```
DELETE FROM
        (SELECT * FROM liniifact lf INNER JOIN facturi f
                ON lf.NrFactf.NrFact AND DataFact = DATE'2007-07-21')
```

va şterge din tabela LINIIFACT toate liniile facturilor emise pe 21 iulie 2007, păstrându-se însă înregistrările din tabela-părinte (FACTURI). Sintaxa este valabilă doar în Oracle.

Dacă, însă, folosim o joncțiune externă, lucrurile se schimbă și în Oracle. Încercăm o altă variantă de ștergere a facturilor fără linii, de data aceasta pe bază unei joncțiuni externe:

```
DELETE FROM
        (SELECT * FROM facturi f LEFT OUTER JOIN liniifact lf
                ON f.NrFact = lf.NrFact AND lf.NrFact IS NULL)
```

Ceea ce obținem este un mesaj de eroare (vezi figura 13.7). De notat că eroarea semnalizată face trimitere la o „vedere" (*view*), subiect abordat abia în capitolul următor.

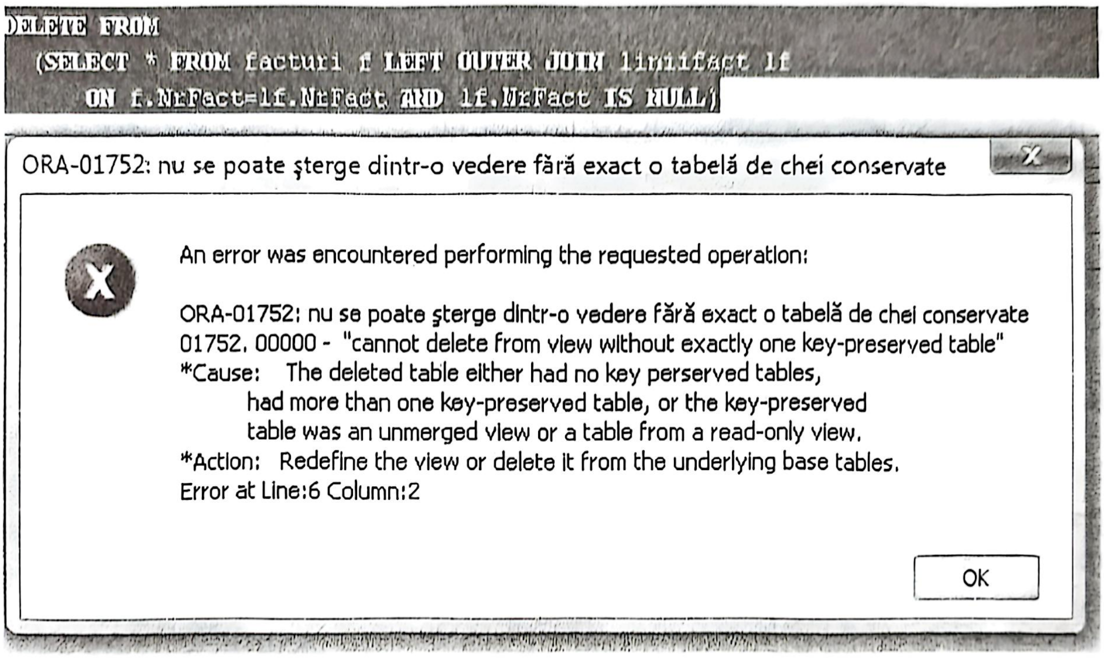


Figura 13.7. Eroare la folosirea joncțiunii externe în clauza FROM a unei comenzi DELETE

Schimbăm un pic registrul, referindu-ne la traseele firmei de transport. Știm din capitolul anterior cum să generăm traseele (rutele) dintre două localități. Să generăm, astfel, o tabelă care va conține, pentru început, traseele Iași-Focșani.

```
CREATE TABLE trasee AS
        SELECT Ord,Traseu,
                (SELECT SUM(Distanta) FROM (
                        SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**'
                                || Loc2 AS Traseu, LEVEL AS Nivel, d.*,
                                ROWNUM AS Ord
                        FROM distante d
                        START WITH Loc1 = 'Iasi'
                        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani'
                                AND Level < 50
                        ORDER BY Ord            ) t2
                WHERE t1.Traseu LIKE Traseu || '%') AS Km
        FROM (
```

```
            SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' ||
                    Loc2 AS Traseu, LEVEL AS Nivel, d.*, ROWNUM AS Ord
            FROM distante d
            START WITH Loc1 = 'Iasi'
            CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani'
                    AND Level < 50
            ORDER BY Ord) t1
    WHERE Traseu LIKE '**Iasi%Focsani'
    ORDER BY Km
```

Acum, că tot le-am creat, a sosit momentul să le ştergem. Dar nu oricum. Ne propunem să eliminăm toate rutele cu distanţa mai mare decât media distanţelor pentru toate rutele Iaşi-Focşani. Nimic mai simplu:

```
DELETE FROM trasee WHERE Traseu LIKE '**Iasi%Focsani' AND Km <
    (SELECT AVG(Km) FROM trasee WHERE Traseu LIKE '**Iasi%Focsani')
```

Redactarea perechii de comenzi CREATE TABLE / DELETE în celelalte dialecte SQL vă rămâne ca temă.

13.3.4. Actualizări şi subconsultări

UPDATE are formatul general apropiat de cel al comenzii DELETE. Faţă de ştergeri, la modificări, lucrurile pot fi mai complicate (implicit, mai interesante), deoarece putem folosi subconsultări atât în predicatul clauzei WHERE, cât şi pentru a preciza expresiile pentru atribuirea valorii unor atribute.

Să începem, însă, cu o problemă mai simplă. Pentru facturile care nu au nici o linie am vrea să adăugăm un „semnal", lipit la eventuala valoare nenulă a atributului Obs:

```
UPDATE facturi
SET Obs = COALESCE(Obs, '') || 'fara linii!!!'
WHERE NrFact NOT IN (SELECT NrFact FROM liniifact)
```

Similar comenzii DELETE, tabela-argument a comenzii UPDATE poate fi înlocuită cu o subconsultare, astfel încât în DB2 şi Oracle este corectă şi versiunea:

```
UPDATE (SELECT Obs FROM facturi WHERE NrFact NOT IN
            (SELECT NrFact FROM liniifact)
    )
SET Obs = COALESCE(Obs, '') || 'fara linii!!!'
```

Pentru a avea o mai mare libertate de mişcare, adăugăm trei atribute pe care le vom lega de denormalizarea bazei de date[3]. Astfel, în tabela FACTURI adăugăm nu mai puţin de cinci atribute:

- *ValTotală* reprezintă valoarea totală (inclusiv TVA) a facturii;

3. Dacă la prima ediţie făceam trimitere la denormalizare păstrând o oarecare discreţie asupra ei, între timp am tratat subiectul ceva mai detaliat în capitolul 9 al cărţii publicate în 2005 [Fotache 2005].

- *TVA* – taxa pe valoarea adăugată pe care ar trebui să o plătim statului (pentru că ne-a dat voie să facem vânzări sau cel puțin aceasta e părerea lui);
- *ValÎncasată* reprezintă suma plătită de client din contravaloarea facturii respective;
- *Reduceri* – dacă un client plătește o factură înainte de termenul obișnuit (să zicem zece zile), îi putem acorda o reducere de 5% pentru că ne-am procopsit rapid cu lichidități; procentul poate fi negociat în mod diferit cu fiecare client;
- *Penalități* este opusul atributului anterior. Prin contract sau prin lege, dacă un client este mai leneș cu plata la timp a facturilor, i se pot aplica penalități, fără a avea însă certitudinea că le vom încasa vreodată.

Practic, prin aceste noi atribute, urmărirea încasării facturilor se schimbă sensibil. *Valoarea de încasat* dintr-o factură este *valoarea totală* plus *penalități* minus *reduceri*. Asta e vestea bună. Vestea proastă ține de faptul că atributul ValTotală n-ar trebui să se modifice decât la actualizarea tabelei LINIIFACT, nu? Altminteri, dacă utilizatorul ar modifica printr-un UPDATE acest atribut, ar apărea un decalaj supărător între liniile din facturi și valorile acestora. Actualizarea automată a unor atribute calculate este unul din scopurile declarate ale declanșatoarelor (*trigger*-elor) pe care le vom discuta în capitolul 17. Adăugăm cele cinci atribute în tabela FACTURI, sintaxa următoare fiind funcțională în toate cele patru dialecte SQL:

```
ALTER TABLE facturi ADD ValTotala NUMERIC(10,2) DEFAULT 0;
ALTER TABLE facturi ADD TVA NUMERIC(10,2) DEFAULT 0;
ALTER TABLE facturi ADD ValIncasata NUMERIC(10,2) DEFAULT 0;
ALTER TABLE facturi ADD Reduceri NUMERIC(9,2) DEFAULT 0;
ALTER TABLE facturi ADD Penalizari NUMERIC(9,2) DEFAULT 0;
```

Acum, dacă tot le-am creat, trebuie să le „umplem" cu valorile curente din tabelele LINIIFACT, PRODUSE și ÎNCASFACT. Pentru calculul valorii totale a unei facturi avem nevoie de o interogare corelată după cum urmează:

```
UPDATE facturi
SET ValTotala = (
            SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
            FROM  liniifact lf INNER JOIN produse p
                    ON lf.CodPr = p.CodPr
            WHERE NrFact = facturi.NrFact
            )
```

Subconsultarea ia în calcul cantitatea, prețul unitar și procentul TVA pentru fiecare produs vândut. Prin corelare se vor lua în considerare numai liniile din LINIIFACT (și PRODUSE) corespunzătoare facturii curente (linia curentă din FACTURI). Și această comandă funcționează fără modificări în toate cele patru servere BD.

În ceea ce privește atributul Reduceri, instituim următoarea regulă: se acordă o reducere de 5% pentru toate tranșele unei facturi încasate în termen de 15 zile de la data vânzării. Iată sintaxa Oracle:

```
UPDATE facturi
SET Reduceri = (
        SELECT SUM (
                CASE WHEN DataInc <= DataFact + INTERVAL '15' DAY
                        THEN Transa * 0.05
                ELSE 0
                END
                        )
        FROM incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
                INNER JOIN facturi f2 ON incf.NrFact = f2.NrFact
        WHERE f2.NrFact = facturi.NrFact
                )[4]
```

sau, mai simplu, folosind corelarea cu liniile din tabela-argument FACTURI (sintaxa valabilă doar în Oracle şi DB2, însă aici trebuie să înlocuim INTERVAL...):

```
UPDATE facturi
SET Reduceri = (
        SELECT  SUM (CASE WHEN DataInc <= DataFact + INTERVAL '15' DAY
                        THEN Transa * 0.05 ELSE 0 END)
        FROM  incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
        WHERE incf.NrFact = facturi.NrFact
                )
```

Atenţie, însă, dacă plasăm condiţia din clauza WHERE în continuarea clauzei ON (ştim din capitolele anterioare că, de obicei, în interogări putem face lucrul acesta fără probleme):

```
UPDATE facturi
SET Reduceri = (
        SELECT  SUM (CASE WHEN DataInc <= DataFact + INTERVAL '15' DAY
                        THEN Transa * 0.05 ELSE 0 END)
        FROM  incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
                AND incf.NrFact = facturi.NrFact
                )
```

chiar în Oracle obţinem un mic mesaj de eroare (vezi figura 13.8), în timp ce DB2-ul nu ridică probleme (dacă înlocuim INTERVAL...).

4. *DataFact + INTERVAL '15' DAY* trebuie înlocuit cu *DataFact + 15 DAY* în DB2, cu *DATEADD(DAY, 15, DataFact)* în SQL Server şi cu *DataFact + INTERVAL '15 DAYS'* în PostgreSQL.

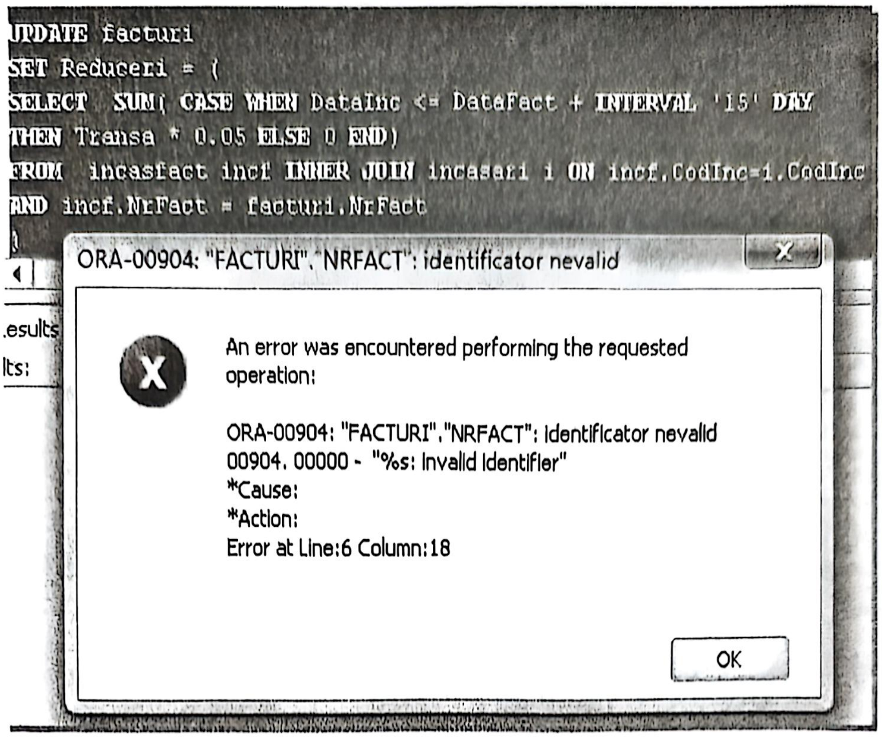


Figura 13.8. Restricție (Oracle) la joncțiunea „corelată" cu tabela „centrală" din UPDATE

Complicăm un pic cazul reducerilor. Acordăm 10% pentru tranșele încasate în mai puțin de 15 zile de la data vânzării, 9% pentru 16 zile și 8% pentru 17 zile. Soluția Oracle este[5]:

```
UPDATE facturi
SET Reduceri = (
        SELECT SUM ( CASE
                WHEN DataInc <= DataFact + INTERVAL '15' DAY THEN Transa * 0.1
                WHEN DataInc <= DataFact + INTERVAL '16' DAY THEN Transa * 0.09
                WHEN DataInc <= DataFact + INTERVAL '17' DAY THEN Transa * 0.08
                ELSE 0
                        END)
        FROM incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
        WHERE incf.NrFact = facturi.NrFact
                )
```

Deși pentru varianta simplificată a reducerilor (interogarea precedentă) nu a ridicat nicio obiecție, la execuția acestui SELECT, SQL Server se împotrivește (vezi figura 13.9).

5. Pentru modificările din celelate dialecte (INTERVAL...), vezi indicațiile de pe pagina anterioară.

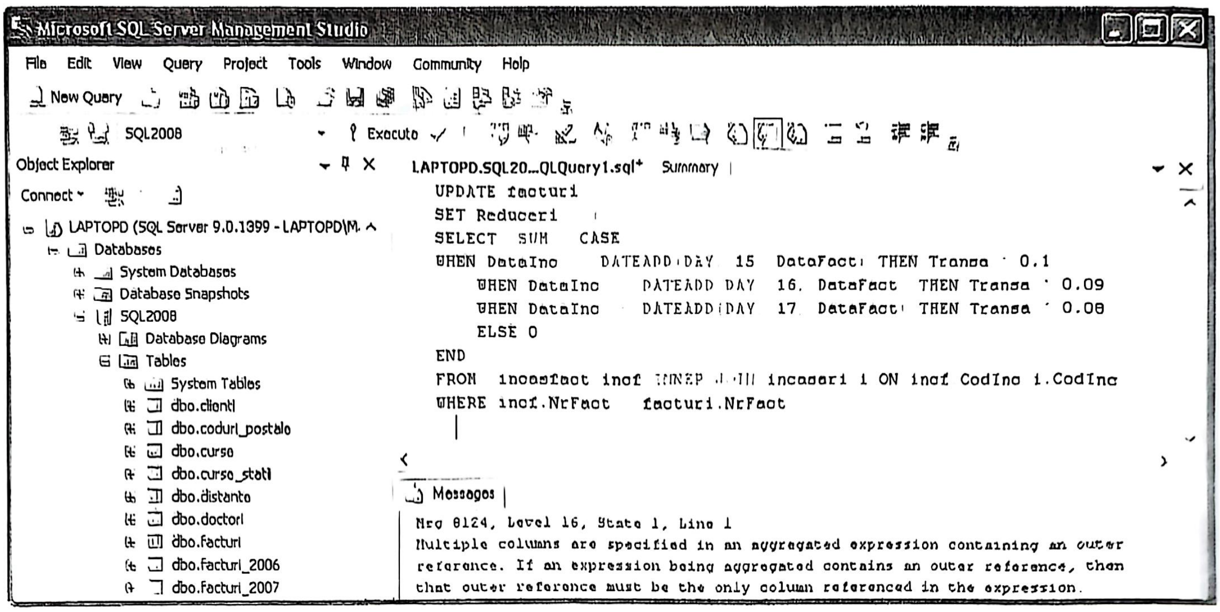


Figura 13.9. Eroare SQL Server cauzată de modificarea structurii CASE

Varianta funcţională este:

```
UPDATE facturi
SET Reduceri = (
        SELECT SUM (
            CASE WHEN DataInc <= DATEADD(DAY, 15, DataFact)
            THEN Transa * 0.1
            ELSE
                CASE WHEN DataInc <= DATEADD(DAY, 16, DataFact)
                THEN Transa * 0.09
                ELSE
                    CASE WHEN DataInc<=
                        DATEADD(DAY, 17, DataFact)
                    THEN Transa * 0.08
                    ELSE 0
                    END
                END
            END)
        FROM incasfact incf
            INNER JOIN incasari i ON incf.CodInc = i.CodInc
            INNER JOIN facturi f2 ON incf.NrFact = f2.NrFact
        WHERE f2.NrFact = facturi.NrFact
                )
```

Întrucât suntem într-o formă samariteană maximă, nu calculăm penalităţi, chiar dacă sunt clienţi care le-ar merita cu vârf şi îndesat. Plasăm cele discutate despre cele cinci atributele calculate într-o interogare „unificatoare":

```
UPDATE facturi
SET
        ValTotala = (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
                    FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
                    WHERE lf.NrFact = facturi.NrFact),
        TVA = (SELECT SUM(Cantitate * PretUnit * ProcTVA)
                FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
                WHERE lf.NrFact = facturi.NrFact),
        ValIncasata = COALESCE((SELECT SUM(Transa)
                            FROM incasfact
                            WHERE NrFact = facturi.NrFact),0),
        Reduceri = COALESCE((SELECT SUM (CASE
                WHEN DataInc <= DataFact + INTERVAL '15' DAY THEN Transa * 0.1
                WHEN DataInc <= DataFact + INTERVAL '16' DAY THEN Transa * 0.09
                WHEN DataInc <= DataFact + INTERVAL '17' DAY THEN Transa * 0.08
                ELSE 0
                                END)
                        FROM incasfact incf INNER JOIN incasari i
                            ON incf.CodInc = i.CodInc
                        WHERE incf.NrFact = facturi.NrFact
                    ),0),
        Penalizari = 0
```

Sintaxa este Oracle, dar cu modificările de rigoare funcționează și în celelalte servere BD. Pentru a vă convinge, în figura 13.10 am listat primele nouă înregistrări ale tabelei FACTURI.

NRFACT	DATAFACT	CODCL	OBS	VALTOTALA	TVA	VALINCASATA	REDUCERI	PENALIZARI
1111	01-08-2007	1001	(null)	4346037,5	687287,5	53996	5399,6	0
1112	01-08-2007	1005	Probleme cu transportul	125516	13116	125516	12551.6	0
1113	01-08-2007	1002	(null)	106275	8775	106275	9864,75	0
1114	01-08-2007	1006	(null)	6021706,5	950856,5	0	0	0
1115	02-08-2007	1001	(null)	151237,5	12487,5	0	0	0
1116	02-08-2007	1007	Pretul propus initial a fost modificat	126712,5	10462,5	0	0	0
1117	03-08-2007	1001	(null)	222050	27050	23204	2320,4	0
1118	04-08-2007	1001	(null)	201975	29475	201975	20197,5	0
1119	07-08-2007	1003	(null)	5774498,5	912848,5	0	0	0

Figura 13.10. Valorile atributelor nou-introduse în FACTURI

Comanda poate fi simplificată, oarecum, prin calcularea simultană a două, respectiv trei atribute folosind câte o singură interogare, însă doar DB2 și Oracle permit acest lucru (noi ne oprim la sintaxa Oracle):

```
UPDATE facturi
SET (ValTotala, TVA, Penalizari) =
            (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)),
```

```
                    SUM(Cantitate * PretUnit * ProcTVA), 0
            FROM liniifact lf
                    INNER JOIN produse p ON lf.CodPr = p.CodPr
            WHERE lf.NrFact = facturi.NrFact),
    (ValIncasata, Reduceri) =
            (SELECT COALESCE(SUM(Transa),0),
            COALESCE((SUM (CASE
            WHEN DataInc <= DataFact + INTERVAL '15' DAY THEN Transa * 0.1
            WHEN DataInc <= DataFact + INTERVAL '16' DAY THEN Transa * 0.09
            WHEN DataInc <= DataFact + INTERVAL '17' DAY THEN Transa * 0.08
            ELSE 0
            END)),0)
            FROM incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
            WHERE incf.NrFact = facturi.NrFact
            )
```

sau

```
UPDATE (SELECT NrFact, DataFact, ValTotala, TVA, ValIncasata, Reduceri, Penalizari
        FROM facturi) f
SET
        (ValTotala, TVA, Penalizari) =
                (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA)),
                        SUM(Cantitate * PretUnit * ProcTVA), 0
                FROM liniifact lf INNER JOIN produse p ON lf.CodPr = p.CodPr
                WHERE lf.NrFact = f.NrFact),
        (ValIncasata, Reduceri) =
                (SELECT COALESCE(SUM(Transa),0),
                    COALESCE((SUM (CASE
                      WHEN DataInc <= DataFact + INTERVAL '15' DAY THEN Transa * 0.1
                      WHEN DataInc <= DataFact + INTERVAL '16' DAY THEN Transa * 0.09
                      WHEN DataInc <= DataFact + INTERVAL '17' DAY THEN Transa * 0.08
                                ELSE 0
                                END)),0)
                FROM incasfact incf INNER JOIN incasari i ON incf.CodInc = i.CodInc
                WHERE incf.NrFact = f.NrFact
                )
```

Trecând prea-puţin-vătămaţi prin experienţa actualizării atributelor calculate în tabela FACTURI, mai adăugăm discret atributul *TVALinie* în LINIIFACT:

```
ALTER TABLE liniifact ADD TVAlinie NUMERIC(9,2) DEFAULT 0;
```

„Umplerea" sa s-ar face destul de simplu cu varianta universală:

```
UPDATE liniifact
```

```
SET TVAlinie = Cantitate * PretUnit *
        (SELECT ProcTVA FROM produse WHERE CodPr = liniifact.CodPr)
```

sau una „particulară" Oracle:

```
UPDATE (SELECT * FROM liniifact INNER JOIN produse
        ON liniifact.CodPr = produse.CodPr)
SET TVALinie = Cantitate * PretUnit * ProcTVA
```

Ca și în cazul comenzii DELETE, Oracle ne oferă o eroare dacă apelăm la o joncțiunea externă (vezi figura 13.11):

```
UPDATE (SELECT Obs FROM facturi LEFT OUTER JOIN liniifact
        ON facturi.NrFact = liniifact.NrFact
        WHERE liniifact.NrFact IS NULL)
SET Obs = COALESCE(Obs, '') || 'fara linii!!!'
```

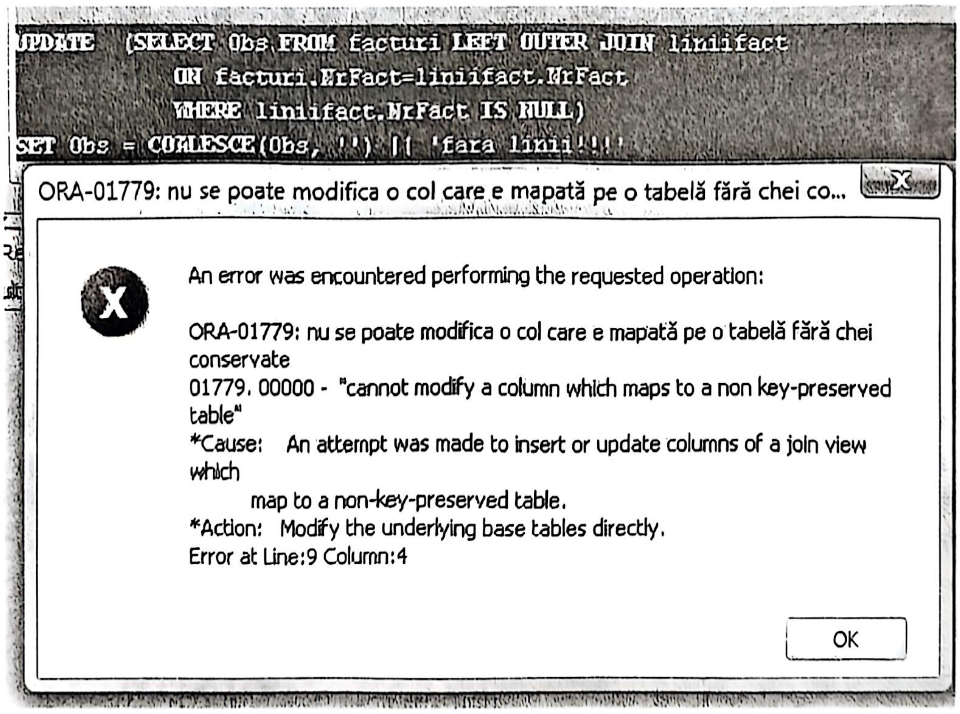


Figura 13.11. Eroare cauzată de folosirea joncțiunii în interogarea-argument a comenzii UPDATE

De data aceasta, nu este vorba numai despre joncțiune externă, ci despre orice tip de joncțiune în care condiția nu se exprimă printr-o cheie candidat (vezi și capitolul următor). Același gen de mesaj de eroare se obține și la execuția comenzii:

```
UPDATE (SELECT Obs FROM facturi INNER JOIN liniifact
        ON facturi.NrFact = liniifact.NrFact)
SET Obs = COALESCE(Obs, '') || 'ARE linii!!!'
```

13.3.5. O comandă recentă: MERGE

Am prezentat comanda MERGE în cartea publicată în 2003, dedicată Oracle ([Fotache *et al.* 2003]), lăsând să se înțeleagă că este o găselniță ceva mai exotică.

Comanda a fost inclusă în standard începând cu SQL:2003, iar dintre cele patru servere BD numai dialectele DB2 şi Oracle au implementat-o. La drept vorbind, apare şi în SQL Server 2008, dar noi am lucrat în această carte preponderent cu versiunea 2005. MERGE este, de fapt, o „combinaţie" INSERT-UPDATE.

Pentru a ilustra discuţia cu un exemplu simplu, să presupunem că utilizatorilor din compartimentul Marketing li se pune la dispoziţie o tabelă redundantă numită CLIENŢI_PRODUSE_ANI cu atributele *DenCl, DenPr, UM, An* şi *Vânzări*, creată astfel:

```
CREATE TABLE clienti_produse_ani AS
        (SELECT DenCl, DenPr, UM, 2007 AS An, 10000000.00 AS Vanzari
        FROM clienti c
            INNER JOIN facturi f ON c.CodCl = f.CodCl
            INNER JOIN liniifact lf ON f.NrFact = lf.Nrfact
            INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE 1 = 2
        )[6];
ALTER TABLE clienti_produse_ani ADD CONSTRAINT pk_cpa
    PRIMARY KEY (DenCl, DenPr, UM, An);
```

Din raţiuni de securitate (şi pentru că cei de la Marketing sunt certaţi cu responsabilii IT), angajaţii compartimentului Marketing au drepturi depline pe această nouă tabelă, însă nu pot consulta tabelele FACTURI şi LINIIFACT, ci numai arhivele lunare ale acestora (de exemplu, FACTURI_2007_1, FACTURI_2007_2 ş.a.m.d), care le parvin după încheierea fiecărei luni. Prin urmare, după fiecare lună *j* din anul *i*, datele din perechea de tabele FACTURI_*i*_*j* - LINIIFACT_*i*_*j* trebuie „vărsate" în CLIENŢI_PRODUSE_ANI.

Dacă vă gândiţi la INSERT-ul:

```
INSERT INTO clienti_produse_ani
        (SELECT DenCl AS DenumireCl, DenPr AS DenumirePr, UM,
                EXTRACT (YEAR FROM DataFact) AS Anul,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Lunare
        FROM clienti c
                INNER JOIN facturi_2007_1 f ON c.CodCl = f.CodCl
                INNER JOIN liniifact_2007_1 lf ON f.NrFact = lf.Nrfact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DenCl, DenPr, UM,  EXTRACT (YEAR FROM DataFact)
        )
```

să ştiţi că nu merge decât pentru ianuarie. În februarie, mai mult ca sigur vom avea vânzări pentru produse/clienţi introduse deja din ianuarie, iar INSERT-ul va încălca restricţia de cheie primară. Nici UPDATE-ul nu e o idee grozavă, pentru că

6. Sintaxă Oracle. În DB2 trebuie adăugat *DEFINITION ONLY*.

pot apărea valori noi ale „cuplului" client-produs. Astfel, cea mai sigură cale e combinația UPDATE/INSERT următoare:

```
— pt. luna august
INSERT INTO clienti_produse_ani
        (SELECT DenCl AS DenumireCl, DenPr AS DenumirePr, UM,
                EXTRACT (YEAR FROM DataFact) AS Anul,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Lunare
        FROM clienti c
                INNER JOIN facturi_2007_8 f ON c.CodCl = f.CodCl
                INNER JOIN liniifact_2007_8 lf ON f.NrFact = lf.Nrfact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DenCl, DenPr, UM,  EXTRACT (YEAR FROM DataFact)
        HAVING (DenCl, DenPr, UM,  EXTRACT (YEAR FROM DataFact)) NOT IN
                        (SELECT DenCl, DenPr, UM, An FROM clienti_produse_ani)
        );

UPDATE clienti_produse_ani SET Vanzari = Vanzari +
        (SELECT SUM(Cantitate * PretUnit * (1 + ProcTVA))
        FROM clienti c
                INNER JOIN facturi_2007_8 f ON c.CodCl = f.CodCl
                INNER JOIN liniifact_2007_8 lf ON f.NrFact = lf.Nrfact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE DenCl = clienti_produse_ani.DenCl AND
                        DenPr = clienti_produse_ani.DenPr
                AND EXTRACT (YEAR FROM DataFact) = clienti_produse_ani.An
        );
```

Ei bine, logica celor două comenzi poate fi înglobală într-o comandă MERGE, după cum urmează:

```
MERGE INTO clienti_produse_ani cpa USING
        (SELECT DenCl AS DenumireCl, DenPr AS DenumirePr, UM,
                EXTRACT (YEAR FROM DataFact) AS Anul,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vanzari_Lunare
        FROM clienti c
                INNER JOIN facturi_2007_1 f ON c.CodCl = f.CodCl
                INNER JOIN liniifact_2007_1 lf ON f.NrFact = lf.Nrfact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        GROUP BY DenCl, DenPr, UM,  EXTRACT (YEAR FROM DataFact)
        ) vanzari_lunare
                ON (cpa.DenCl = vanzari_lunare.DenumireCl AND
                    cpa.DenPr = vanzari_lunare.DenumirePr AND
```

```
            cpa.UM = vanzari_lunare.UM AND cpa.An = vanzari_lunare.Anul)
WHEN MATCHED THEN
      UPDATE
      SET Vanzari = Vanzari + vanzari_lunare.Vanzari_Lunare
WHEN NOT MATCHED THEN
      INSERT
      VALUES (vanzari_lunare.DenumireCl, vanzari_lunare.DenumirePr,
      vanzari_lunare.UM, vanzari_lunare.Anul, vanzari_lunare.Vanzari_Lunare);
```

Tabela specificată după clauza INTO este cea în care vor fi efectuate actualizările - în cazul nostru CLIENȚI_PRODUSE_ANI (sinonim local CPA). După clauza USING este plasată fraza SELECT care centralizează vânzările pentru fiecare combinație client-produs-unitate de măsură-an. Comanda UPDATE de pe ramura WHEN MATCHED se va executa numai pentru valorile combinației din fraza SELECT care există, la momentul lansării comenzii, în CLIENȚI_PRODUSE_ANI. Valorile combinației ce nu se găsesc în tabela destinație vor fi adăugate tabelei prin intermediul comenzii INSERT de pe ramura WHEN NOT MATCHED.

Capitolul 14
Tabele temporare şi virtuale

În capitolul anterior am prezentat câteva exemple de tabele create şi populate prin SELECT-uri mai mult sau mai puţin complicate. O bună parte dintre acestea ne-au folosit la stocarea temporară a unor rezultate de formă tabelară, rezultate ce ar putea constitui sursa unor rapoarte sau chiar a unor controale din formulare. Pentru rapoarte sau popularea controalelor de tip listă, combo, mult mai nimerită este folosirea tabelelor temporare şi/sau virtuale. Acestea, spre deosebire de tabelele „obişnuite", nu ocupă spaţiu pe disc, decât pe parcursul unei sesiuni de lucru sau chiar mai puţin, întrucât în schema bazei se stochează permanent doar structura lor, nu şi conţinutul.

14.1. Tabele temporare

Tabelele temporare se creează prin aceeaşi comandă - CREATE TABLE cu cele două variante majore - listă de atribute şi/sau restricţii sau consultări (SELECT). Există două tipuri de tabele temporare ([Melton & Simon 2002], pp. 100-102), *locale* şi *globale*. Conţinutul ambelor tipuri nu este vizibil decât utilizatorului curent (sesiunii curente, pentru că între sesiunile de lucru conţinutul unei tabele temporare poate fi stocat doar prin copierea într-o tabelă obişnuită). Gulutzan şi Pelzer le numesc *tabele-dependente-de-sesiune* ([Gulutzan & Pelzer 1999], p. 350).

Tabelele temporare locale sunt cele mai „invizibile", conţinutul lor fiind disponibil doar înăuntrul aceluiaşi modul de cod executat în cadrul unei sesiuni de lucru. Doar cele globale pot fi „transmise" între modulele executate într-o aceeaşi sesiune de lucru, însă şi aici trebuie avut grijă, pentru că tabelele temporare pot *păstra* conţinutul lor la finalizarea unei tranzacţii (ON COMMIT PRESERVE ROWS), dar îl pot şi pierde automat (ON COMMIT DELETE ROWS). Cele două clauze funcţionează şi pentru tabelele virtuale locale.

Prima utilizare majoră a tabelelor temporare vizează simplificarea interogărilor. Ne referim, pentru început, la frustrarea pe care au trăit-o utilizatorii de PostgreSQL în paragraful 9.6 dedicat expresiilor-tabelă (WITH... SELECT...), paragraf în care numai PostgreSQL-iştii au stat pe tuşă.

Să luăm problema atât de îndrăgită despre bere: *Care este judeţul în care berea s-a vândut cel mai bine?* Folosind o tabelă temporară, şi în acest dialect lucrurile pot fi simplificate sensibil:

```
CREATE LOCAL TEMPORARY TABLE judete_bere AS
        (SELECT Judet, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Vnz_Bere
```

```
        FROM judete j
                INNER JOIN coduri_postale cp ON j.Jud = cp.Jud
                INNER JOIN clienti c ON cp.CodPost = c.CodPost
                INNER JOIN facturi f ON c.CodCl = f.CodCl
                INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                INNER JOIN produse p ON lf.CodPr = p.CodPr
        WHERE Grupa = 'Bere'
        GROUP BY Judet) ;

SELECT * FROM judete_bere
WHERE Vnz_Bere = (SELECT MAX(Vnz_Bere) FROM judete_bere);
```

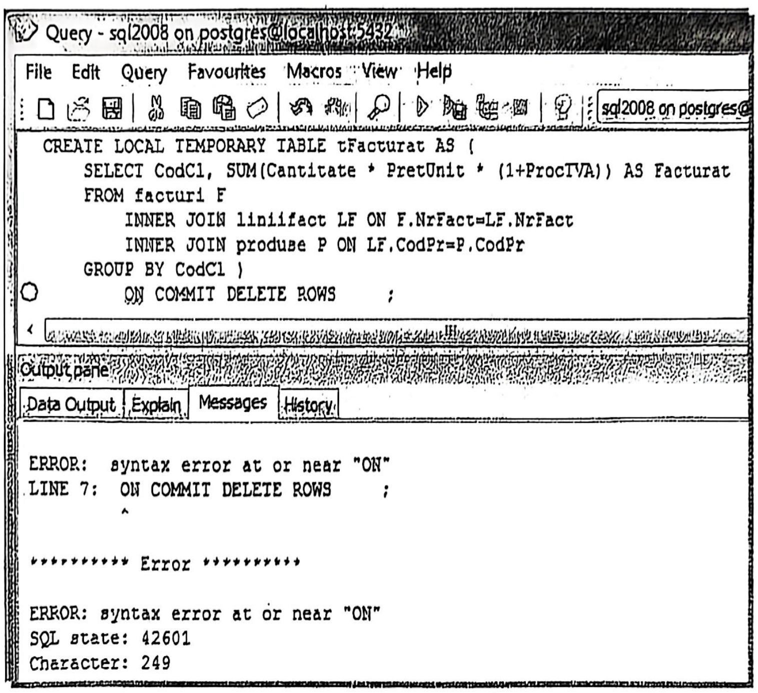


Figura 14.1. O (altă) tulburare de personalitate a PostgreSQL-ului

Practic, expresia-tabelă a devenit tabelă temporară. Acum, dacă tot a venit vremea confidenţelor, trebuie să vă mărturisesc că PostgreSQL-ul „iese din frontul" standardului, deoarece:

- nu păstrează definiţia tabelelor temporare între sesiuni;
- nu face diferenţierea între tabelele temporare globale şi locale;
- dacă, în lipsa clauzei ON COMMIT, standardul prevede ştergerea liniilor, în PostgreSQL este invers – când lipseşte ON COMMIT, conţinutul se păstrează;
- clauza ON COMMIT are, pe lângă cele două opţiuni standard – PRESERVE ROWS şi DELETE ROWS –, clauza DROP care semnalizează că, la finalul tranzacţiei, tabela urmează să fie ştearsă.

În plus (de fapt, în minus!), nu am reuşit să folosesc clauza ON COMMIT, mesajul de eroare fiind, de fiecare dată de genul celui din figura 14.1[1].

1. Vezi şi documentaţia PostgreSQL la pagina *http://www.postgresql.org/docs/8.3/interactive/sql-createtable.html.*

Care sunt cei mai mari trei 3 datornici ?

În varianta „scripturală" PostgreSQL, pe care o vom reda mai jos, vom crea trei tabele temporare locale/globale: *tFACTURAT* care conține vânzările pentru fiecare client, *tÎNCASAT* ce conține totalul încasărilor de la fiecare client şi *tDE_INCASAT*, cu restul de plată al fiecărui client:

```
CREATE LOCAL TEMPORARY TABLE tFacturat AS (
        SELECT CodCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat
        FROM facturi F
                INNER JOIN liniifact LF ON F.NrFact = LF.NrFact
                INNER JOIN produse P ON LF.CodPr = P.CodPr
        GROUP BY CodCl;
CREATE LOCAL TEMPORARY TABLE tIncasat AS
        SELECT CodCl, SUM(Transa) AS Incasat
        FROM facturi F INNER JOIN incasfact I ON F.NrFact = I.NrFact
        GROUP BY CodCl;
CREATE LOCAL TEMPORARY TABLE tDe_INCASAT AS
        SELECT DenCl, tFACTURAT.CodCl, Facturat,
                COALESCE(Incasat,0) AS Incasat,
                Facturat - COALESCE(Incasat,0) AS De_Incasat
        FROM tFACTURAT
        INNER JOIN clienti ON tFacturat.CodCl = clienti.CodCl
                LEFT OUTER JOIN  Tincasat
                        ON  tFACTURAT.CodCl = tINCASAT.CodCl;
```

Interogarea finală face apel doar la ultima tabelă temporară:

```
SELECT * FROM tDe_INCASAT WHERE De_Incasat > =
        (SELECT MAX(De_Incasat) FROM tDe_INCASAT WHERE De_Incasat <
            (SELECT MAX(De_Incasat) FROM tDe_INCASAT WHERE De_Incasat <
                (SELECT MAX(De_Incasat)  FROM tDe_INCASAT)
        )   );
```

Fiind siliți de sintaxa rigidă a funcției CONNECTBY(), în paragraful 12.5 am creat o tabelă virtuală - *PERSONAL2_MODIF01*. Puteam la fel de bine folosi şi o tabelă temporară în locul celei virtuale. Analog, simplificăm obținerea traseelor Iaşi-Focşani (şi distanței fiecărei rute - vezi figura 12.33), dacă recurgem la o tabelă temporară - X.

```
CREATE LOCAL TEMPORARY TABLE x AS
        SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta, ierarhie.Cale
        FROM (SELECT *
                FROM CONNECTBY('distante', 'Loc2', 'Loc1', 'Iasi', 0, ' *  * ')
                        AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
                ) ierarhie, distante d
        WHERE ierarhie.Loc2 = d.Loc1 AND ierarhie.Loc1 = d.Loc2 ;
```

```
SELECT x1.cale, SUM(x2.distanta) AS Km
FROM x x1, x x2
WHERE x1.cale LIKE 'Iasi%Focsani' AND x1.cale LIKE x2.cale || '%'
GROUP BY x1.cale
ORDER BY Km
```

În capitolul 17 vom vedea la lucru tabelele temporare PostgreSQL, dar în altă ipostază – cea de substitut al variabilelor publice.

Şi în Oracle apar câteva diferenţe faţă de „evangheliile" SQL în materie de tabele temporare. Poate cea mai importantă este că în Oracle nu pot fi create tabele temporare locale, ci numai globale. Încercăm să simplificăm interogarea din paragraful 13.1 care acum produce rezultatul din figura 13.4:

```
CREATE GLOBAL TEMPORARY TABLE tab ON COMMIT PRESERVE ROWS AS
        SELECT SYS_CONNECT_BY_PATH (Loc1, '* * ') || '* * ' || Loc2 AS Traseu,
                LEVEL AS Nivel, Loc1, Loc2,
                EXTRACT (HOUR FROM Durata) * 60 + EXTRACT (MINUTE
                        FROM durata) AS Durata_Min, ROWNUM AS Ord
        FROM distante d START WITH Loc1 = 'Iasi'
        CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
        ORDER BY Ord;


SELECT 10001 AS IdCursa, 1 AS "StatieNr", 'Iasi' AS "Statie",
        TIMESTAMP'2008-04-01 09:05:00' AS "Data/Ora"
FROM dual
        UNION
SELECT 10001, t2.Nivel + 1, t2.Loc2, TIMESTAMP'2008-04-01 09:05:00' +
        TO_DSINTERVAL ('0 ' || FLOOR(
(SELECT SUM(Durata_Min) FROM tab WHERE t2.Traseu LIKE Traseu || '%' )/
                60) || ':' || MOD ( (SELECT SUM( Durata_Min )
                                FROM tab WHERE t2.Traseu
                                LIKE Traseu || '%' ), 60)
                || ':00')
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '* * Iasi * * Vaslui%Focsani'
        AND t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2
```

Interesant este şi modul în care ştergem o tabelă temporară în Oracle. Astfel, comanda DROP generează, fără o „pregătire" prealabilă, mesajul de eroare din figura 14.2.

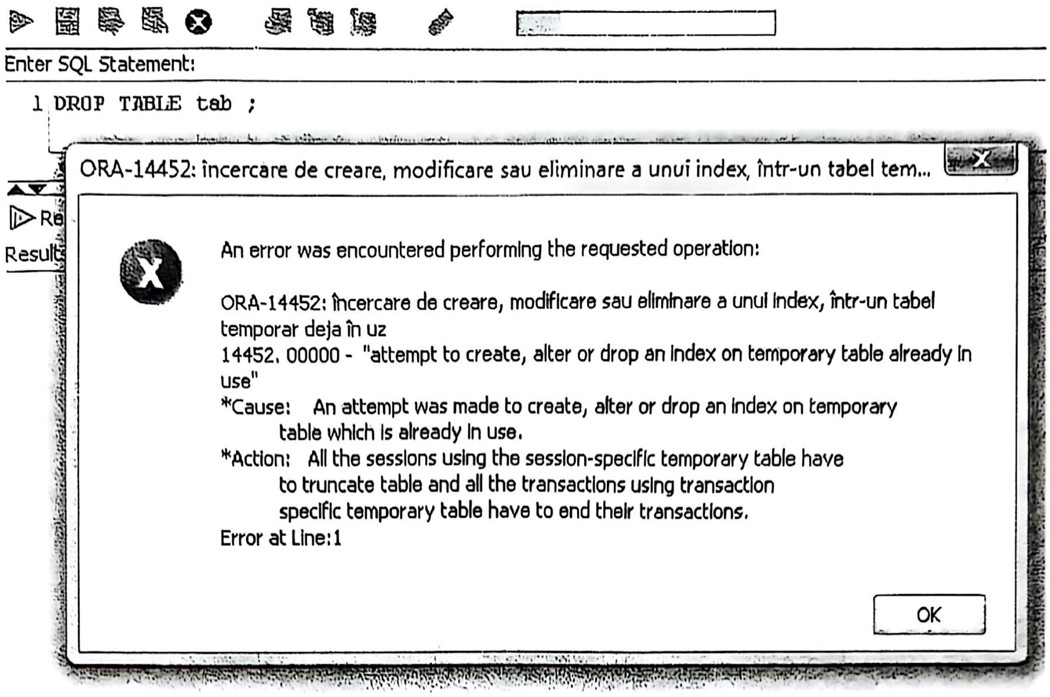


Figura 14.2. Tentativă eşuată de ştergere a unei tabele temporare în Oracle

Din mesaj ar rezulta că tabela temporară ar trebui, mai întâi, ştearsă cu TRUNC. Într-adevăr, prin succesiunea celor două comenzi:

```
TRUNCATE TABLE tab;
DROP TABLE tab;
```

ştergerea decurge fără probleme.

DB2 are (ca şi SQL Server, dar în altă direcţie) o opinie diferită în materie de tabele temporare. În DB2, o tabelă temporară este definită la nivel de sesiune, însă descrierea sa nu apare în dicţionarul de date (catalogul de sistem). De aceea, o tabelă temporară nu poate fi partajată cu alte sesiuni. La terminarea sesiunii, orice urmă a tabelei temporare dispare. Comanda nu este CREATE TABLE, ci DECLARE GLOBAL TEMPORARY TABLE care este însă destul de limitată. Copiem atributele tabelei CURSE din paragraful 13.1 în tabela temporară tCURSE:

```
DECLARE GLOBAL TEMPORARY TABLE tCurse AS (
        SELECT 1001 AS IdCursa, CAST (' ' AS VARCHAR(200)) AS Traseu,
        CAST ('2008-04-01 09:05:00' AS TIMESTAMP) AS DataOra_Plecare
FROM sysibm.sysdummy1) DEFINITION ONLY ;
```

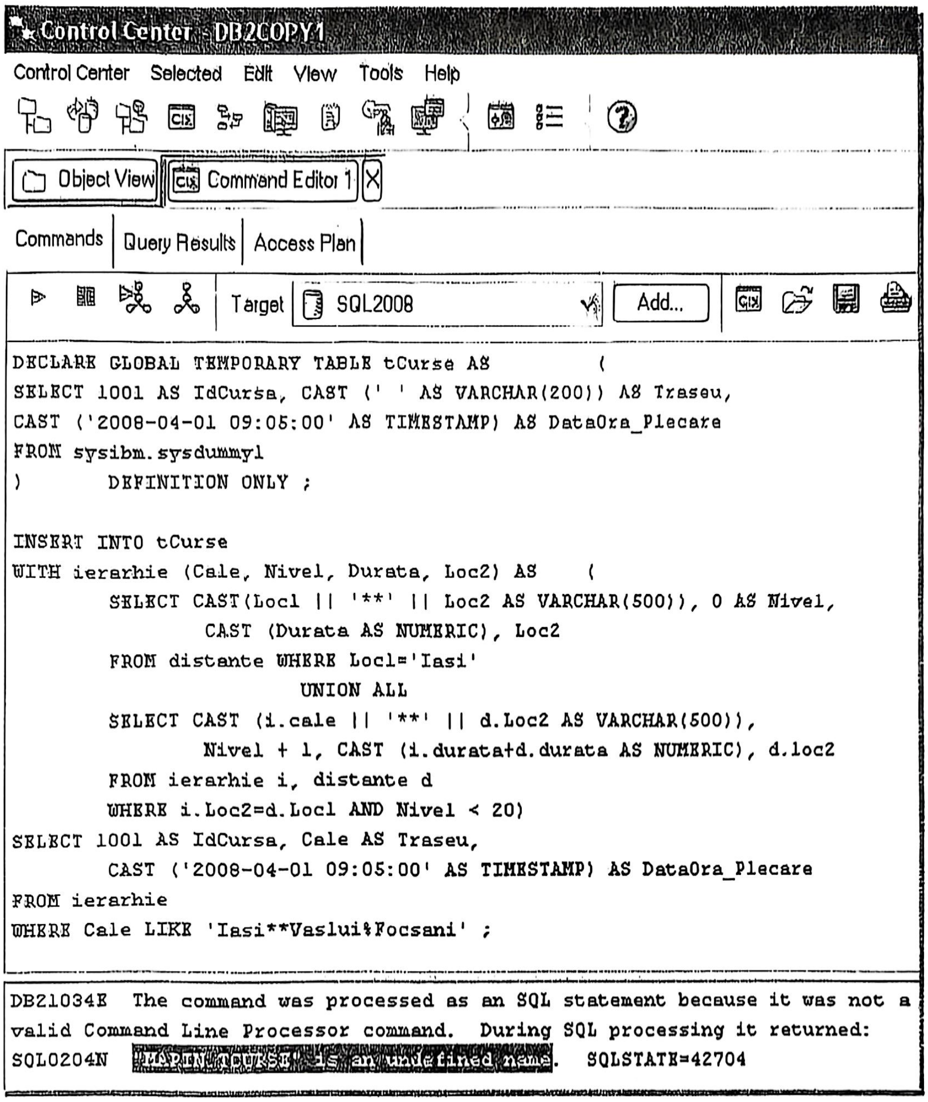


Figura 14.3. Tabele „extrem de" temporare în DB2

În schimb, când încercăm să populăm tabela temporară cu INSERT-ul din paragraful 13.3.1 avem parte de un mesaj destul de ciudat, potrivit căruia *tCURSE* este un obiect necunoscut (un fel de OZN) (vezi figura 14.3). Explicaţia este că DB2-ul uită extrem de repede definiţia tabelei temporare. Fireşte că vă întrebaţi, ca şi mine, la ce sunt bune tabelele temporare în DB2. Răspunsul va fi oferit în următoarea ediţie a cărţii (care, dacă ar urma trendul acesteia, va apărea prin 2015).

Şi SQL Server este departe de sintaxa din standardul SQL. Cele două tipuri de tabele temporare sunt specificate prin simboluri „#" plasate înaintea numelui. Astfel, dacă, la creare, numele tabelei este prefixat de un singur simbol „#", atunci tabela temporară este una locală, vizibilă deci numai în sesiunea curentă. Dacă prefixul numelui tabelei este format din două semne „#" (##), atunci tabela temporară este globală. Reluăm exemplul celor trei datornici prezentat şi pentru PostgreSQL:

```
SELECT CodCl, SUM(Cantitate * PretUnit * (1 + ProcTVA)) AS Facturat
INTO #tFacturat
FROM facturi F INNER JOIN liniifact LF ON F.NrFact = LF.NrFact
        INNER JOIN produse P ON LF.CodPr = P.CodPr
```

```
GROUP BY CodCl;
SELECT CodCl, SUM(Transa) AS Incasat
INTO #tIncasat
FROM facturi F INNER JOIN incasfact I ON F.NrFact = I.NrFact
GROUP BY CodCl;
SELECT DenCl, #tFacturat.CodCl, Facturat, COALESCE(Incasat,0) AS Incasat,
        Facturat - COALESCE(Incasat,0) AS De_Incasat
INTO #tDe_INCASAT
FROM #tFacturat
        INNER JOIN clienti ON #tFacturat.CodCl = clienti.CodCl
        LEFT OUTER JOIN #tIncasat ON #tFacturat.CodCl = #tIncasat.CodCl;

SELECT * FROM #tDe_INCASAT WHERE De_Incasat > =
(SELECT MAX(De_Incasat) FROM #tDe_INCASAT WHERE De_Incasat <
        (SELECT MAX(De_Incasat) FROM #tDe_INCASAT WHERE De_Incasat <
                (SELECT MAX(De_Incasat) FROM #tDe_INCASAT)
        ) );
```

14.2. Tabele virtuale în interogări

Asemenea celor temporare, tabelele virtuale (*view*-urile) sunt construite prin subconsultări aplicate asupra tabelelor de bază şi/sau altor tabele temporare sau *view*-uri. Tot similar celor temporare, numai definiţia (structura) este persistentă şi publică (accesibilă altor sesiuni/utilizatori), conţinutul nu! Există însă şi diferenţe majore. O tabelă virtuală nu are conţinut propriu-zis, fiind „instanţiată" doar la invocarea acesteia (invocare ce înseamnă, de fapt, execuţia interogării-definiţie). Dar cea mai importantă deosebire este că numai o parte dintre view-uri sunt actualizabile, iar modificarea lor se propagă automat în tabelele-sursă. Inserarea, modificarea sau ştergerea unei linii dintr-un *view actualizabil* se va traduce, în fapt, prin inserarea, modificarea sau ştergerea de linii în/din una sau mai multe dintre tabelele-sursă.

Avantajele tabelelor virtuale sunt multiple:

- simplificarea interogărilor, prin salvarea rezultatelor intermediare;
- construirea unor surse de date pentru rapoarte şi controale ale formularelor (liste, combo-uri sau chiar grid-uri);
- facilitarea vizualizării informaţiilor „presărate" în multe tabele;
- mai bună securitate a datelor; anumitor categorii de utilizatori, în locul accesului la tabelele bazei, li se poate acorda acces exclusiv la view-uri în care pot vedea datele pe care sunt autorizaţi să le consulte/modifice.

Standardul SQL-92 a introdus view-urile ca tabele virtuale ce sunt materializate la invocarea numelui lor. Materializarea înseamnă execuţia frazei SELECT ce constituie definiţia tabelei virtuale şi popularea cu înregistrările-rezultat ale interogării. Formatul simplist al comenzii de creare a unei tabele virtuale este:

```
CREATE VIEW <nume-tabelă-virtuală> [<listă-coloane>] AS
        <frază SELECT>
        [WITH [<clauză de nivele>] CHECK OPTION]
```

unde *<clauză de nivele>* :: = *CASCADED | LOCAL*.

Odată creată tabela virtuală, definiţia sa este salvată în schema bazei. Ulterior, ori de câte ori este necesar, la deschiderea/reîmprospătarea tabelei virtuale, aceasta este (re)populată cu înregistrări extrase din cele ale tabelelor propriu-zise ce apar în clauza FROM a interogării. O tabelă virtuală poate fi aşadar privită şi ca o expresie de subconsultare a unei tabele persistente, stocată în bază şi invocată prin numele său.

Potrivit standardelor SQL, unei tabele virtuale nu i se pot asocia indexuri şi nici nu i se pot impune restricţii, însă unele SGBD-uri optimizează lucrul cu *view*-urile folosind indexurile tabelelor persistente. Numele tabelelor virtuale este unic şi nu se poate autoreferi, deşi un *view* poate fi creat pe baza unei combinaţii de tabele persistente şi/sau alte *view*-uri.

Ştergerea unei tabele virtuale din schema bazei de date se realizează prin comanda DROP VIEW:

```
DROP VIEW <nume-tabelă-virtuală> <comportament>
```

unde *<comportament>* :: = *CASCADE | RESTRICT*. Prin clauza CASCADE, se şterg atât tabela virtuală curentă, cât şi toate cele create pe baza sa, în timp ce RESTRICT interzice operaţiunea, atâta timp cât există măcar un view construit pe baza tabelei virtuale curente.

De regulă, formatul general al comenzii de creare a tabelelor virtuale este mai generos decât cel al unei tabele temporare. Spre exemplu, în DB2 la definirea unei tabele temporare nu putem include o expresie-tabelă, în timp ce la o tabelă virtuală da. Comanda următoare defineşte o tabelă virtuală, NUMERE_NATURALE, ce va conţine numere naturale (atributul Număr) de la 0 la 9999:

```
CREATE VIEW numere_naturale AS
      WITH templ (Numar) AS (
            VALUES (0) UNION ALL
            SELECT Numar + 1 FROM templ WHERE Numar < 10000
                        )
            SELECT * FROM templ;
```

Reformulăm, astfel, o problemă pentru care am formulat soluţia în paragraful 12.6: *Care sunt numerele de facturi nefolosite (în tabela FACTURI)?*

```
SELECT Numar AS Numar_nefolosit
FROM (SELECT Numar FROM numere_naturale WHERE Numar BETWEEN
            (SELECT MIN(NrFact) FROM facturi) AND
                  (SELECT MAX(NrFact) FROM facturi )
      ) LEFT OUTER JOIN facturi f ON Numar = NrFact
WHERE f.Nrfact IS NULL
ORDER BY 1
```

Tabelele virtuale sunt preferabile celor temporare şi datorită modului de reîmprospătare a conţinutului. În cadrul aceleiaşi sesiuni, o tabelă temporară globală nu-şi schimbă conţinutul (dacă a fost definită cu clauza ON COMMIT PRESERVE ROWS). Dacă sesiunea este lungă, este probabil ca tabelele din care provine tabela temporară să fi fost actualizate. Modificările sunt reflectate automat la următoarea invocare a view-ului, în timp ce tabela temporară va fi reîmprospătată abia la următoarea iniţializare (următoarea sesiune).

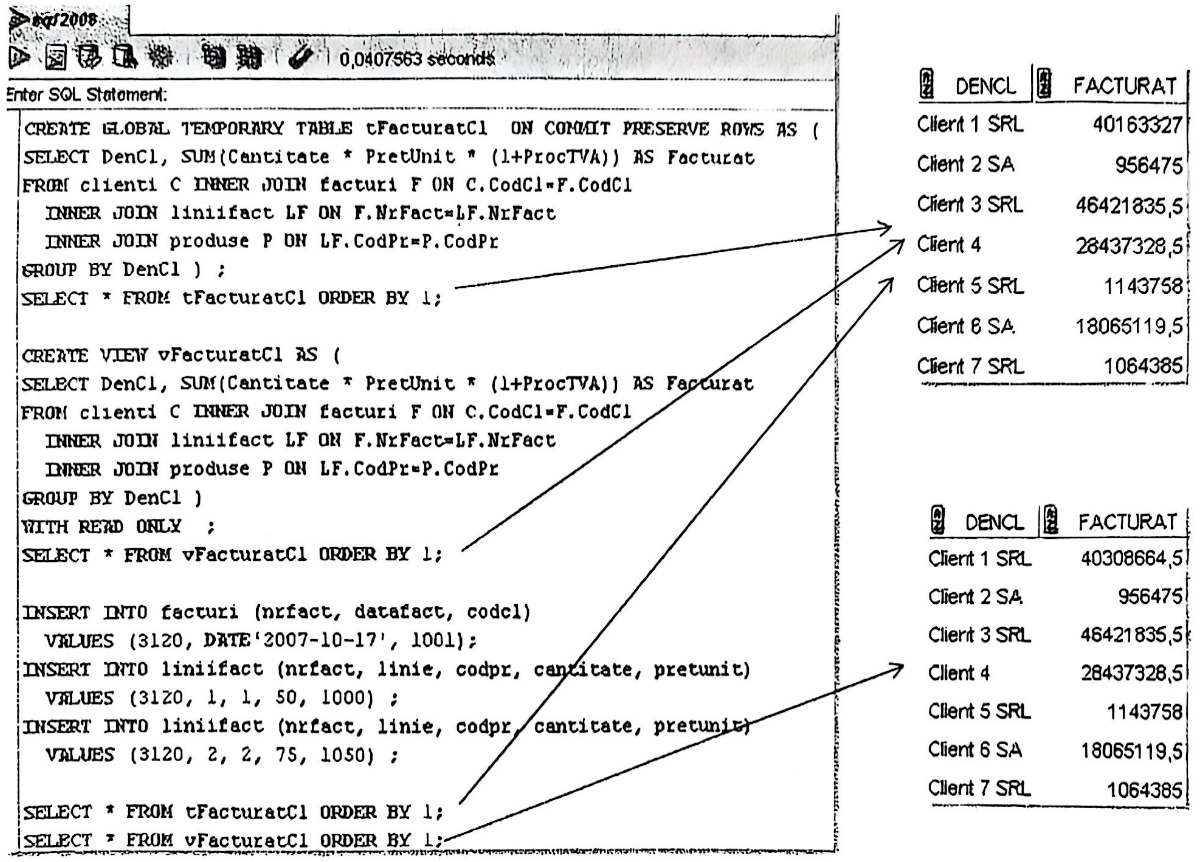


Figura 14.4. Actualizarea dinamică a tabelelor virtuale din tabelele-sursă

Figura 14.4 prezintă o sesiune în care sunt create o tabelă temporară (*tFacturatCl*) şi una virtuală (vFacturatCl) care, iniţial, au conţinut identic. În aceeaşi sesiune, se adaugă o factură (3120) cu două linii pentru *Clientul 1 SQL*. După inserare, conţinutul tabelei temporare este identic cu cel de după crearea sa, în timp ce prima linie din tabela virtuală se împrospătează cu noua valoare a atributului *Facturat* pentru *Client 1 SRL*.

14.3. Probleme ale actualizărilor tabelelor-sursă pe baza modificării tabelelor virtuale

Am văzut în figura 14.4 că o tabelă virtuală se împrospătează automat cu modificările operate în tabelele-sursă. Ei bine, în unele cazuri şi reciproca este valabilă, adică o serie de view-uri sunt modificabile (din punctul de vedere al

conţinutului), iar modificările se propagă în tabela sau tabelele-sursă. Numai că aici există o serie întreagă de restricţii. Pentru *vFacturatCl*, discuţia se simplifică din start, deoarece la creare s-a folosit opţiunea READ ONLY, ceea ce înseamnă că orice tentativă de inserare, modificare şi ştergere va fi automat tratată cu duşmănie.

Pentru a fi actualizabilă, fiecare înregistrare a unei tabele virtuale trebuie să poată fi asociată unei singure linii dintr-o tabelă-sursă. Astfel, modificarea unei linii din *view* poate fi propagată fără probleme de ambiguitate. Fireşte, o tabelă derivată de genul:

```
CREATE VIEW vJudete1 AS
        SELECT * FROM Judete
        WHERE regiune = 'Moldova' OR regiune = 'Dobrogea'
```

nu va crea probleme insolubile la actualizare. În schimb, deşi cu un conţinut identic *vJudeţe1*, tabela virtuală *vJudeţe2*, creată prin comanda care urmează, nu este actualizabilă nici în SQL, nici în majoritatea SGBD-urilor importante.

```
CREATE VIEW vJudete2 AS
        SELECT * FROM Judete WHERE regiune = 'Moldova'
                UNION
        SELECT * FROM Judete WHERE regiune = 'Dobrogea'
```

Pentru a intra în câteva detalii, să imaginăm o tabelă derivată denumită ÎNCA-SĂRI_CLIENŢI (figura 14.5), ce conţine documentul de încasare, suma încasată (corespunzătoare documentului) şi clientul care a efectuat plata. Comanda de creare a tabelei virtuale este:

```
CREATE VIEW vINCASARI_CLIENTI AS (
SELECT DenCl, CodDoc, NrDoc, DataDoc, SUM(Transa) AS SumaPlatita
FROM clienti c
        INNER JOIN facturi f ON c.CodCl = f.CodCl
        INNER JOIN incasfact incf ON f.NrFact = incf.NrFact
        INNER JOIN incasari inc ON incf.CodInc = inc.CodInc
GROUP BY DenCl, CodDoc, NrDoc, DataDoc
);
```

DENCL	CODDOC	NRDOC	DATADOC	SUMAPLATITA
Client 1 SRL	OP	333	09-08-2007	117069
Client 1 SRL	CEC	444	10-08-2007	9754
Client 1 SRL	OP	111	10-08-2007	155971
Client 2 SA	OP	555	10-08-2007	106275
Client 1 SRL	OP	666	11-08-2007	3696
Client 5 SRL	CHIT	222	15-08-2007	125516

Figura 14.5. O tabelă virtuală

Utilitatea unei asemenea relaţii virtuale poate fi certificată de un angajat al compartimentului financiar care se ocupă, printre altele, şi de evidenţa încasărilor de la

clienţi. Pe cât de utilă, pe atât de problematică devine ÎNCASĂRI_CLIENŢI atunci când se pune problema actualizării sale şi propagării modificărilor în tabelele de bază din care provine, CLIENŢI, FACTURI, ÎNCASĂRI şi ÎNCASFACT. Iată câteva dintre probleme:

- modificarea atributului *SumaPlătită* este imposibil de operat în tabela de bază, ÎNCASFACT. O linie din tabela virtuală corespunde uneia sau mai multor linii din tabela de bază (depinde câte facturi sunt achitate prin documentul de plată respectiv). Dacă pentru ordinul de plată 111 din 10 august 2007 se doreşte modificarea (corecţia) sumei din *155 971* în *156 000*, nu se poate cunoaşte tranşa şi factura unde s-a comis eroarea şi, normal, unde trebuie operată modificarea;
- modificarea celui de-al patrulea tuplu din (*Client 2 SA, OP, 555, 10-Aug-2007, 106275*) în (*Client 5 SRL, OP, 555, 10-Aug-2007, 106275*), adică modificarea clientului pentru Ordinul de plată nr. 555 din 10.08.2007, poate fi operată în trei moduri în tabele de bază:
  - fie se modifică în FACTURI pentru factura 1113 valoarea atributului *CodCl* din *1002* în *1005*,
  - fie se modifică în linia din ÎNCASFACT codul încasării (*CodInc*) din *1238* în *1235*,
  - fie se modifică în ÎNCASFACT valoarea atributului *NrFact* din *1113* în *1112*, păstrându-se codul încasării neschimbat,
  - chiar dacă nu la fel de plauzibile, cele trei variante generează o „stare" de confuzie pentru SGBD;
- inserarea unei linii în view este o acţiune temerară, dar fără prea mulţi sorţi de izbândă: oricare ar fi cele patru tabele sursă în care s-ar face inserarea, cel puţin un atribut important (ce nu poate avea valori NULL) nu are valori specificate, astfel încât se încalcă una dintre restricţii (de entitate sau comportament). Spre exemplu, convenim că inserarea trebuie să se propage numai în ÎNCASFACT. Nu se cunoaşte însă numărul facturii încasate. Cum *NrFact* este o componentă a cheii primare a tabelei, este clar că operaţiunea va fi interzisă de SGBD;
- ştergerea unei linii din tabela virtuală ridică problema identificării liniei sau liniilor dintr-una sau mai multe tabele de bază.

În plus, sursele unei tabele virtuale pot fi atât tabele propriu-zise, cât şi alte tabele virtuale, situaţie în care vorbim despre mai multe nivele ale tabelelor virtuale. Spre exemplu, pentru a construi o tabelă virtuală *vCLIENTI* în care pot fi modificate: denumirea clientului, adresa sa, denumirea localităţii în care îşi are sediul şi numele judeţului, sunt necesare următoarele view-uri:

```
CREATE VIEW vJudete AS
        SELECT * FROM Judete;
CREATE VIEW vCoduri_Postale AS
        SELECT CodPost, Loc, vJudete.Jud, Judet, Regiune
        FROM coduri_postale cp INNER JOIN vJUDETE ON cp.Jud = vJUDETE.Jud;
CREATE VIEW vClienti AS
        SELECT CodCl, DenCl, Adresa, vCoduri_Postale.CodPost, Loc, Jud,
```

```
        Judet, Regiune
    FROM clienti INNER JOIN vCoduri_Postale
            ON clienti.CodPost = vCoduri_Postale.CodPost;
```

Tabelele virtuale trebuie să includă toate coloanele cheilor primare/alternative ale tabelelor de bază. O linie dintr-o tabelă virtuală trebuie să corespundă unei singure linii din tabela de bază. Toate coloanele neincluse în view trebuie să admită valori NULL sau să prezinte valori implicite. Altminteri, inserarea unei linii într-o tabelă derivată ar fi imposibilă. Nu poate fi actualizată o tabelă virtuală creată prin fraze SELECT în care apar:
- clauze GROUP BY/HAVING;
- funcţii-agregat;
- coloane calculate;
- operatorii: UNION, INTERSECT, EXCEPT (MINUS);
- SELECT DISTINCT.

SQL:1999 defineşte câţiva termeni legaţi de tabelele virtuale sau consultările-argument ale comenzilor de actualizare ([Melton & Simon 2002], pp. 111-112, 283-284):
- potenţial actualizabil;
- actualizabil;
- banal-actualizabil;
- inserabil-în.

O specificaţie de interogare (care poate fi expresia de definire a unui view sau o expresie ce urmează comenzii UPDATE - vezi paragraful 13.3.4) este *potenţial actualizabilă* dacă şi numai dacă ([Melton & Simon 2002], pp. 111-112, 284):
- nu se foloseşte clauza DISTINCT;
- toate coloanele din clauza SELECT apar în listă o singură dată;
- nu se folosesc clauze GROUP BY/HAVING.

Dacă specificaţia de interogare este potenţial actualizabilă, iar în clauza FROM apare o singură tabelă, atunci toate coloanele din această tabelă care apar în specificaţie sunt *actualizabile*. Dacă specificaţia este potenţial actualizabilă, însă în clauza FROM apar două sau mai multe tabele, atunci o coloană din specificaţie este actualizabilă numai dacă:
- provine dintr-o singură tabelă dintre cele din clauza FROM;
- între tabela respectivă şi specificaţia de interogare există o relaţie de corespondenţă a unicităţii, adică tabela îşi păstrează în interogare cheia primară sau cheile alternative.

Altfel spus, valoarea unui atribut dintr-o tabelă virtuală este actualizabilă doar dacă aceasta corespunde unei singure valori dintr-o tabelă-sursă. O specificaţie (expresie) de interogare este actualizabilă dacă măcar una dintre coloane este actualizabilă şi *banal-actualizabilă* dacă:
- este actualizabilă;
- clauza FROM conţine o singură tabelă;

- toate coloanele sale sunt actualizabile.

Termenul (destul de nenorocit) *inserabil-în* desemnează o specificaţie/expresie actualizabilă, dacă toate tabelele din clauza FROM acceptă adăugări de linii (INSERT-uri) şi dacă expresia nu include operatorii UNION, INTERSECT şi EXCEPT.

SGBD-urile actuale prezintă câteva diferenţe în materie de reguli privind actualizarea tabelelor virtuale. Cel mai flexibil mecanism este, însă, cel al declanşatoarelor de tip *INSTEAD OF*, despre care vom discuta pe scurt în capitolul 17.

14.3.1. Tabele virtuale în DB2

Documentaţia IBM şi bibliografia DB2 (vezi de exemplu [Baklarz & Zikopoulos 2007]) delimitează tabelele virtuale DB2 în funcţie de operaţiunile pe care le acceptă, după cum urmează:
- tabele virtuale ce permit ştergerea (*deletable*);
- tabele virtuale ce permit modificarea (*updatable*);
- tabele virtuale ce permit inserarea (*insertable*);
- tabele virtuale ce permit doar citirea, nemodificabile (*read-only*);

Cele din prima categorie acceptă comanda DELETE, comandă pe baza căreia ştergerea se propagă într-o tabelă-sursă. Pentru a putea propaga ştergerea, fraza SELECT de creare a tabelei virtuale nu trebuie să conţină clauze precum DISTINCT, GROUP BY/HAVING, VALUES, UNION, INTERSECT, EXCEPT, iar fiecare linie din *view* corespunde unei singure linii dintr-o tabelă-sursă. Singurul operator ansamblist acceptabil în unele situaţii este UNION ALL. Un view este *actualizabil* (*updatable*) dacă, în plus, are măcar un atribut a cărui valoare poate fi modificată şi *inserabil* (*insertable*), dacă toate coloanele sale sunt *actualizabile*.

Astfel, *vClienţi*, creată ceva mai sus, nu permite ştergerea unei linii (vezi figura 14.6). Cum nici măcar ştergerile nu sunt admise, cu atât mai puţin modificările şi inserările.

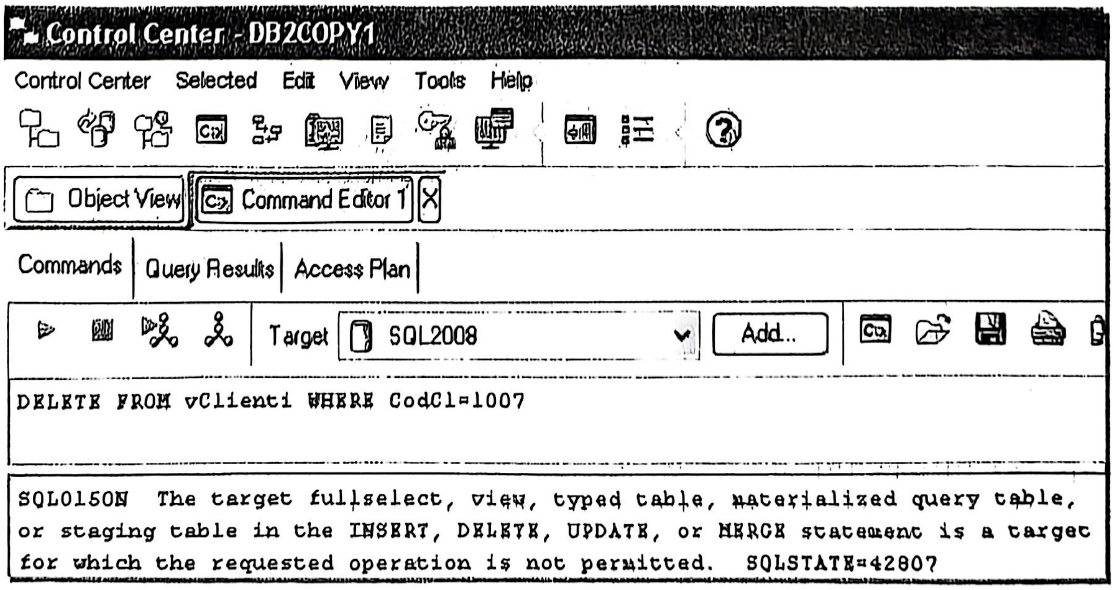


Figura 14.6. O tabelă virtuală din care nu pot fi şterse linii

Tabela virtuală *vLiniiFact7001*, fiind definită printr-o selecţie aplicată unei singure tabele - LINIIFACT:

```
CREATE VIEW vLiniifact7001 AS
        SELECT NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact WHERE NrFact = 7001 ;
```

acceptă orice gen de modificare, fiind *inserabilă*, suportă oricare dintre comenzile următoare:

```
DELETE FROM vLiniifact7001 WHERE Linie = 3;
UPDATE vLiniifact7001 SET Cantitate = 100 WHERE Linie = 2;
INSERT INTO vLiniifact7001 VALUES (7001, 3, 4, 30, 730);
```

Nici tabela virtuală *vLiniifact7001_7002* nu acceptă ştergerea de înregistrări, deşi operatorul folosit este UNION ALL:

```
CREATE VIEW vLiniifact7001_7002 AS
        SELECT * FROM liniifact WHERE NrFact = 7001
        UNION ALL
        SELECT * FROM liniifact WHERE NrFact = 7002;
```

Comanda următoare se va solda cu un mesaj de eroare similar celui din figura 14.6, deoarece, pentru a funcţiona, UNION ALL trebuie să conecteze tabele-sursă diferite:

```
DELETE FROM vLiniifact7001_7002 WHERE NrFact = 7001 AND Linie = 3
```

14.3.2. Tabele virtuale în Oracle

Sintaxa Oracle pentru definirea tabelelor virtuale este un pic mai generoasă decât cea din DB2. Dacă în DB2, pentru modificarea frazei SELECT ce defineşte un view aveam nevoie de o combinaţie DROP VIEW/CREATE VIEW, în Oracle se poate folosi CREATE OR REPLACE VIEW. De asemenea, în DB2, o tabelă virtuală era READ ONLY în funcţie de modul său de definire. În Oracle se poate folosi clauza READ ONLY pentru a bloca modificarea conţinutului unui *view* actualizabil.

```
CREATE OR REPLACE VIEW vLiniifact7001 AS
        SELECT NrFact, Linie, CodPr, Cantitate, PretUnit
        FROM liniifact WHERE NrFact = 7001;
```

Regulile privind actualizarea conţinutului unei tabele virtuale în vederea propagării în tabelele-sursă sunt apropiate de cele din DB2, aşa că nu mai insistăm.

14.3.3. Tabele virtuale în PostgreSQL

PostgreSQL are, probabil, cea mai săracă sintaxă (dintre cele patru dialecte) în materie de CREATE VIEW:

```
CREATE VIEW vJudete AS
        SELECT * FROM judete ORDER BY Judet;
```

După creare, un *view* este destul de apatic în PostgreSQL. Astfel, dacă se încearcă inserarea unei noi înregistrări corespunzătoare judeţului Galaţi, mesajul este cel din figura 14.7. Asta înseamnă că toate tabelele virtuale sunt READ ONLY în PostgreSQL. Totuşi, din explicaţii deducem că trebuie să apelăm la mecanismul de reguli.

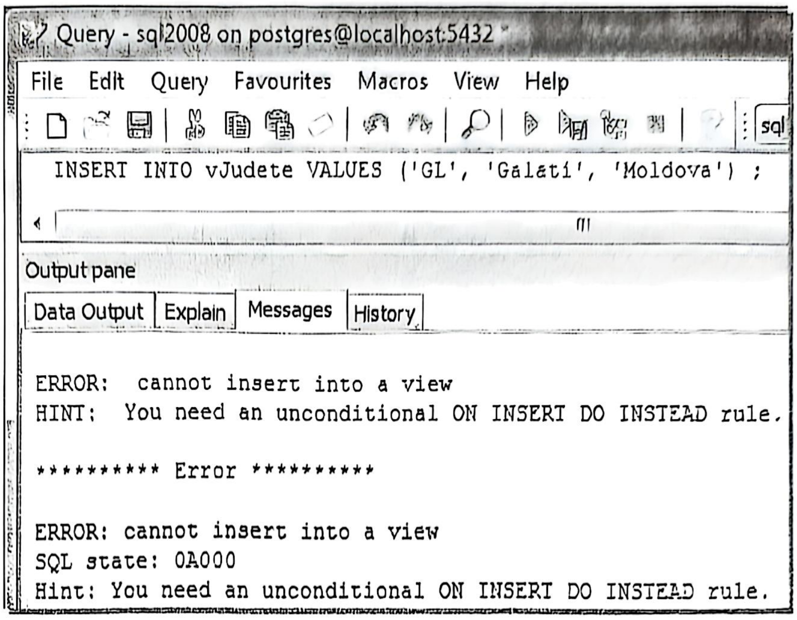


Figura 14.7. Probleme PostgreSQL chiar şi la inserări într-o tabelă virtuală banal-actualizabilă

Acest mecanism de reguli prefaţează destul de bine scurta discuţie din capitolul 17 dedicată declanşatoarelor de tip INSTEAD OF. Tabela virtuală *vJudeţe* este creată dintr-o singură tabelă, JUDEŢE, aşa că inserarea unei linii în view *trebuie* să atragă după sine adăugarea înregistrării respective în tabelă:

```
CREATE RULE vJudete_ins AS ON INSERT TO vJudete DO INSTEAD
        INSERT INTO Judete VALUES (NEW.Jud, NEW.Judet, NEW.Regiune) ;
```

Numele fiecărui atribut este prefixat în clauza VALUES cu *NEW.*, ceea ce semnalizează că este vorba despre valorile atributelor de pe noua linie a tabelei virtuale. Acum, comanda INSERT funcţionează:

```
INSERT INTO vJudete VALUES ('GL', 'Galati', 'Moldova');
```

La modificarea unei înregistrări, vom folosi şi clauza *OLD.* pentru a califica valoarea dinaintea eventualei modificări a atributului cheie primară - Jud.

```
CREATE RULE vJudete_upd AS ON UPDATE TO vJudete DO INSTEAD
        UPDATE judete
        SET Jud = NEW.Jud, Judet = NEW.Judet, Regiune = NEW.Regiune
        WHERE Jud = OLD.Jud;
```

Redactarea regulii pentru ştergere devine acum o treabă simplă:

```
CREATE RULE vJudete_del AS ON DELETE TO vJudete
        DO INSTEAD DELETE FROM judete WHERE Jud = OLD.Jud;
```

Fireşte, modificarea unei valori din *view* care încalcă restricţii definite în tabela de bază va genera un mesaj de eroare (vezi figura 14.8).

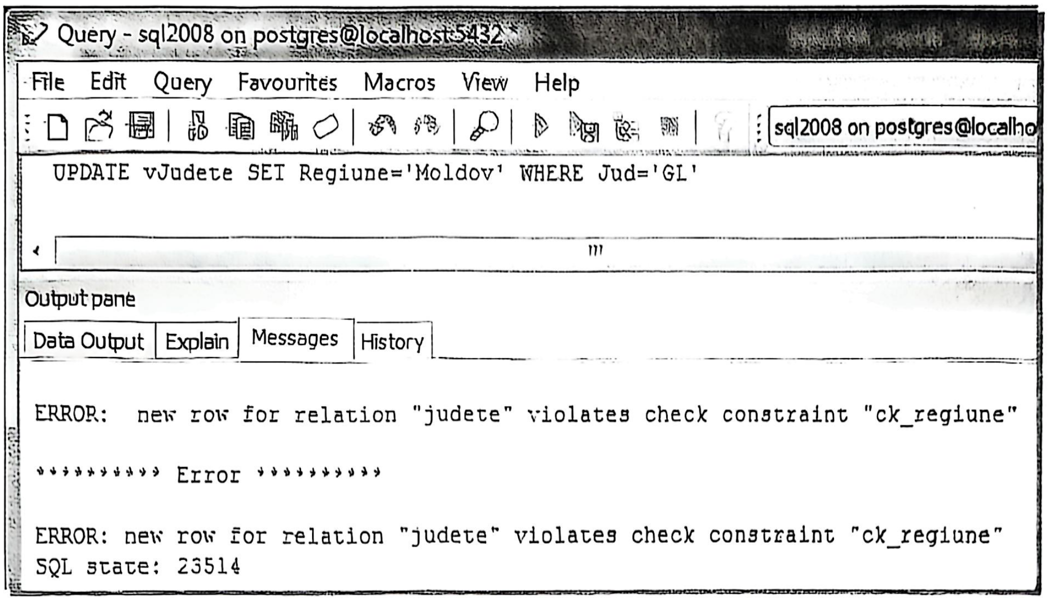


Figura 14.8. Încălcarea unei restricţii la inserarea într-o tabelă virtuală

Mecanismul regulilor ne permite să încercăm rezolvarea problemelor de actualizare chiar şi pentru tabele virtuale cu două sau mai multe tabele-sursă joncţionate. Iată cazul view-ului *vFacturi*:

```
CREATE VIEW vFacturi AS
    SELECT lf.NrFact, DataFact, DenCl, Linie, lf.CodPr, DenPr, UM,
        ProcTVA, Grupa,
        Cantitate, PretUnit, Cantitate * PretUnit AS ValFaraTVALinie,
        Cantitate * PretUnit * ProcTVA AS TVALinie,
        Cantitate * PretUnit * (1 + ProcTVA) AS ValTotLinie
    FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
    ORDER BY lf.NrFact, Linie;
```

Deşi sunt nu mai puţin de patru tabele-sursă, tabela „cea mai de jos" este LINIIFACT, aşa că suntem tentaţi să scriem regula de inserare astfel:

```
CREATE OR REPLACE RULE vFacturi_ins AS
    ON INSERT TO vFacturi DO INSTEAD
    INSERT INTO liniifact VALUES ( NEW.NrFact, NEW.Linie,
        NEW.CodPr, NEW.Cantitate, NEW.PretUnit );
```

Regula funcţionează dacă adăugăm o linie unei facturi existente, ca de exemplu a cincea linie din factura 3119:

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit)
  VALUES (3119, 5, 6, 1000, 1200) ;
```

Dar însă adăugăm o linie într-o factură nouă - să zicem factura cu nr. 3120 -, fireşte că vom încălca restricţia referenţială (vezi figura 14.9).

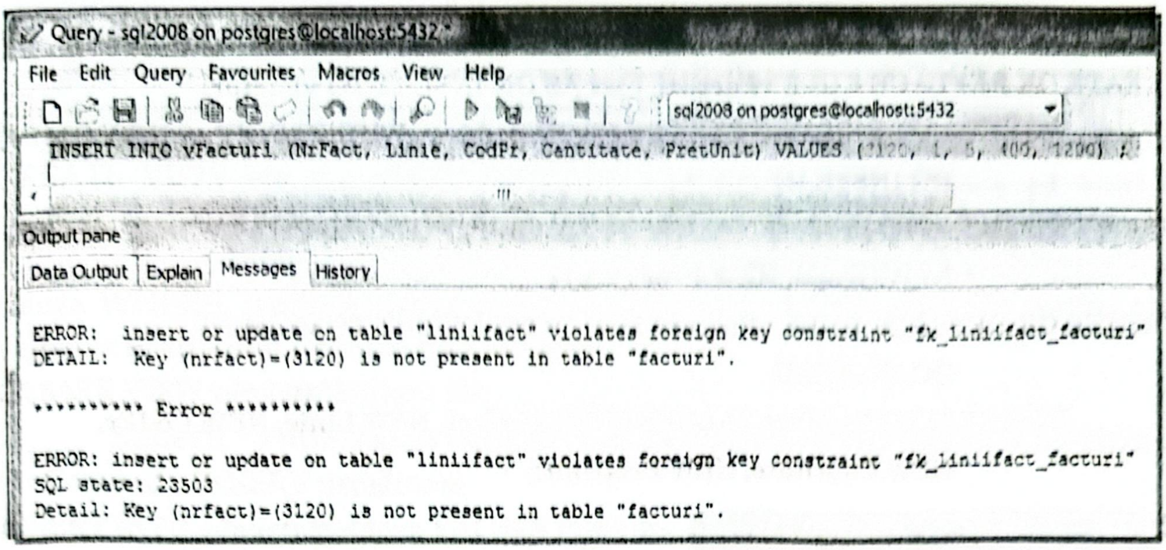


Figura 14.9. Încălcarea restricţiei referenţiale la inserarea într-un view cu mai multe tabele--sursă

Avem soluţii şi pentru acest gen de probleme. Să redefinim tabela virtuală, preluând şi codul clientului:

```
DROP VIEW vFacturi;
CREATE VIEW vFacturi AS
        SELECT lf.NrFact, DataFact, DenCl, f.CodCl, Linie, lf.CodPr, DenPr,
                UM, ProcTVA, Grupa,
                Cantitate, PretUnit, Cantitate * PretUnit AS ValFaraTVALinie,
                Cantitate * PretUnit * ProcTVA AS TVALinie,
                Cantitate * PretUnit * (1 + ProcTVA) AS ValTotLinie
        FROM liniifact lf
                INNER JOIN produse p ON lf.CodPr = p.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
                INNER JOIN clienti c ON f.CodCl = c.CodCl
        ORDER BY lf.NrFact, Linie;
```

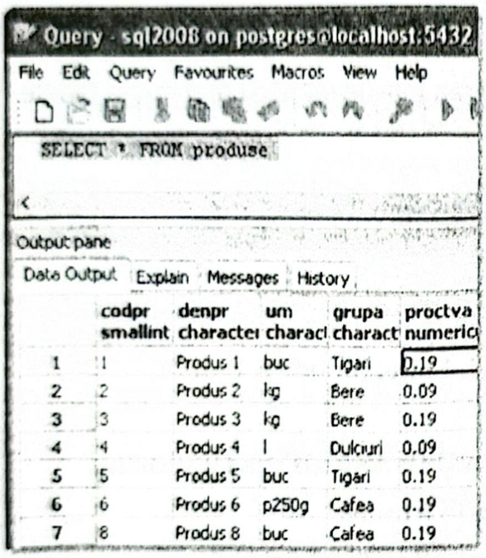
Problema cea mai delicată la rescrierea regulii de inserare este că un INSERT în *vFacturi* poate să se propage în tabele-sursă în mai multe moduri:

- inserarea unei linii numai în tabela LINIIFACT;
- inserarea a câte unei linii în tabelele LINIIFACT şi FACTURI;
- inserarea câte unei linii în tabelele LINIIFACT şi PRODUSE;
- inserarea a câte unei linii în tabelele LINIIFACT, FACTURI şi PRODUSE.

Din fericire, putem formula mai multe reguli pentru aceeaşi operaţiune (inserare), astfel:

```
DROP RULE IF EXISTS vFacturi_ins ON vFacturi;
CREATE OR REPLACE RULE vFacturi_ins1 AS ON INSERT TO vFacturi
        WHERE NOT EXISTS (SELECT 1 FROM facturi WHERE NrFact = NEW.NrFact
                DO INSTEAD
```

```
        INSERT INTO facturi VALUES (NEW.NrFact, NEW.DataFact, NEW.CodCl);
CREATE OR REPLACE RULE vFacturi_ins2 AS ON INSERT TO vFacturi
        WHERE NOT EXISTS (SELECT 1 FROM produse WHERE CodPr = NEW.CodPr)
                DO INSTEAD
        INSERT INTO produse VALUES (NEW.CodPr, NEW.DenPr, NEW.UM,
                NEW.Grupa, NEW.ProcTVA ) ;
CREATE OR REPLACE RULE vFacturi_ins3 AS ON INSERT TO vFacturi
                DO INSTEAD
        INSERT INTO liniifact VALUES ( NEW.NrFact, NEW.Linie, NEW.CodPr,
                NEW.Cantitate, NEW.PretUnit);
```



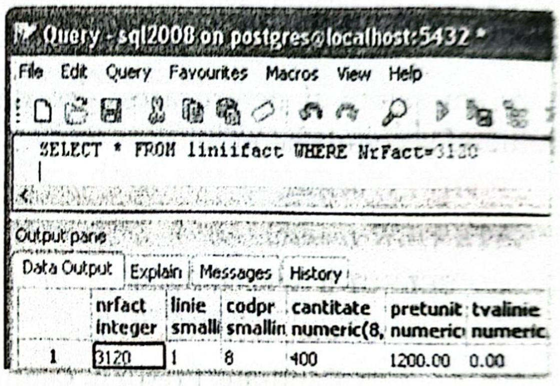


Figura 14.10. Propagarea inserării în tabelele PRODUSE şi LINIIFACT

Testarea funcţionalităţii o vom face printr-un INSERT în care vom adăuga o linie dintr-o factură nouă şi un produs la fel de nou:

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
                CodCl, DenPr, UM, Grupa, ProcTVA)
VALUES (3120, 1, 8, 400, 1200, DATE'2007-10-01', 1001, 'Produs 8', 'buc', 'Cafea', 0.19);
```

Figura 14.10 demonstrează că s-au făcut inserările în cele trei tabele (în tabela FACTURI, inserarea este implicită, altminteri restricţia referenţială ar fi fost încălcată).

14.3.4. Tabele virtuale în SQL Server

SQL Server este comparabil în materie de tabele virtuale cu dialectele din DB2 şi Oracle. Comenzile rulate în DB2 descrise în paragraful 14.3.1 sunt funcţionale şi în SQL Server, aşa că nu mai insistăm.

14.4. Restricţii în tabele virtuale

După cum am văzut la începutul paragrafului 14.2, formatul general al comenzii CREATE VIEW are şi o clauză importantă pe care o vom discuta pe scurt în continuare - WITH CHECK OPTION. Această clauză permite ca la orice inserare sau modificare, înregistrările din tabela virtuală să respecte predicatul formulat în clauza WHERE. *vJudeţeMoldova* conţine înregistrările tabelei JUDEŢE pentru care valoarea atributului Regiune este *Moldova*:

CREATE VIEW vJudeteMoldova AS

SELECT * FROM judete WHERE Regiune = 'Moldova' WITH CHECK OPTION

Comanda INSERT următoare:

INSERT INTO vJudeteMoldova VALUES ('BC', 'Bacau', 'Moldova')

se execută, fără probleme, ceea ce nu este valabil şi pentru (vezi figura 14.11):

INSERT INTO vJudeteMoldova VALUES ('AR', 'Arad', 'Banat')

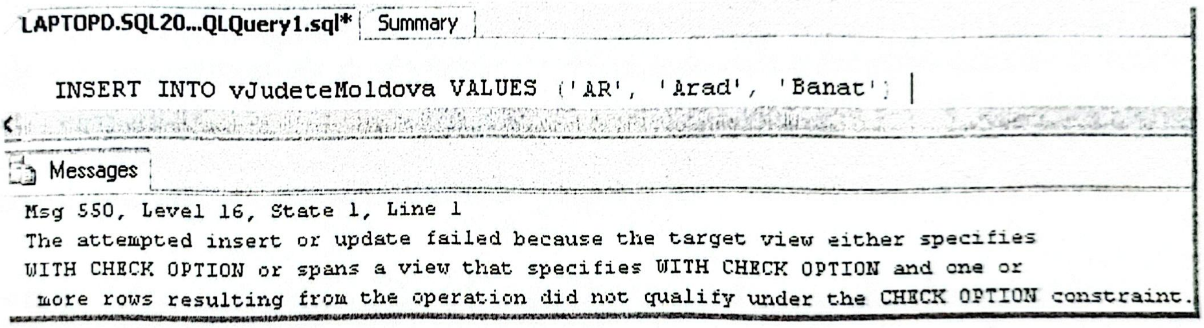


Figura 14.11. Blocarea unei inserări de către clauza WITH CHECK OPTION

Dacă sursa unei tabele virtuale este tot o tabelă virtuală care, la rândul său, este tot o tabelă virtuală ş.a.m.d., clauza CHECK OPTION poate folosi opţiunea CASCADED pentru a verifica dacă inserarea/modificarea respectă predicatele din clauzele WHERE ale tuturor view-urilor (de pe toate nivelele) sau LOCAL pentru a verifica numai respectarea predicatului din clauza WHERE a tabelei virtuale curente. Dintre cele patru dialecte SQL, numai PostgreSQL nu are implementată clauza WITH CHECK OPTION, aşa că şi pentru acest gen de restricţii sunt necesare declanşatoare sau reguli.

Standardele SQL nu prevăd declararea de restricţii „clasice" (cheie primară, cheie alternativă, restricţie referenţială etc.) într-o tabelă virtuală. Nici dialectele DB2, SQL Server şi PostgreSQL. Oracle permite trei tipuri de restricţii declarabile pentru un *view*: chei primare, chei alternative şi restricţii referenţiale.

Capitolul 15
Scheme, obiecte şi utilizatori

Până acum am lăsat să se înţeleagă că suntem singurii utilizatori ai bazei de date. În realitate, am creat obiecte ale bazei de date într-o (sub)schemă în care am avut voie să ne facem de cap. Noţiunea de *subschemă* din acest capitol vine în completarea şi, pe alocuri, în contradicţie cu noţiunea de *schemă* formulată în capitolul 2. Spuneam atunci că schema se referă la *structura* bazei (tabele, tabele virtuale, restricţii etc). În continuare, schema este alcătuită din subscheme, fiecare subschemă fiind accesibilă unuia sau mai multor utilizatori/grupuri.

Dat fiind că în aplicaţiile integrate, bazele de date pot fi organizate în mai multe subscheme, utilizatorii şi grupurile de utilizatori având drepturi diferite la aceste subscheme, ne interesează cum le putem crea, cum acordăm drepturi la obiecte ale subschemelor altor utilizatori/grupuri decât cei „titulari", şi cum răsfoim dicţionarul de date pentru a afla informaţii despre toate obiectele dintr-o (sub)schemă.

15.1. Crearea de subscheme şi acordarea de drepturi

Termenul *subschemă* este, aşadar, acum mai apropiat de *container* sau de *director* (cu semnificaţia din sistemele de operare, de „folder" de fişiere), fiind o colecţie de obiecte. Începând cu SQL-92, sintaxa SQL pentru crearea schemei este:

CREATE SCHEMA <clauză de denumire>

[<specificarea setului de caractere>] [<element>...]

unde:

<clauză de denumire> : : = = <nume schemă> | AUTHORIZATION

<identificator de autorizare>

| <nume schemă> AUTHORIZATION < identificator de autorizare>

<specificarea setului de caractere> : : = =

DEFAULT CHARACTER SET <specificatorul setului de caractere>

<element> : : = = <definiţie domeniu>

| <definiţie tabelă> | <definiţie tabelă virtuală>

| <comandă pentru acordare de drepturi> | <definiţie aserţiune>

Sintaxa şi regimul de funcţionare ale subschemelor, inclusiv mecanismul de creare a utilizatorilor şi de acordare/revocare de drepturi la obiecte din fiecare subschemă, diferă în bună măsură de la un SGBD la altul. Mai trebuie adăugat că diferenţele de titulatură şi organizare ale subschemelor ţin de specificul serverelor, inclusiv de cel al instalării.

15.1.1. Utilizatori şi subscheme în DB2

În DB2, la crearea unei scheme, aceasta poate fi „repartizată" unui utilizator (titular). Suntem interesaţi să creăm un utilizator cu numele *salarizare* pentru care vom defini o schemă cu acelaşi nume. Din păcate, utilizatorii bazei pot fi creaţi doar interactiv, întrucât dialectul SQL nu conţine comanda CREATE USER. Din meniul arborescent al *Control Center* (vezi figura 15.1), după selectarea opţiunii *DB Users* din meniul arborescent (stânga figurii) se afişează meniul contextual din care se alege *Add User* (meniul contextual se afişează printr-un click pe butonul din dreapta al mouse-ului.)

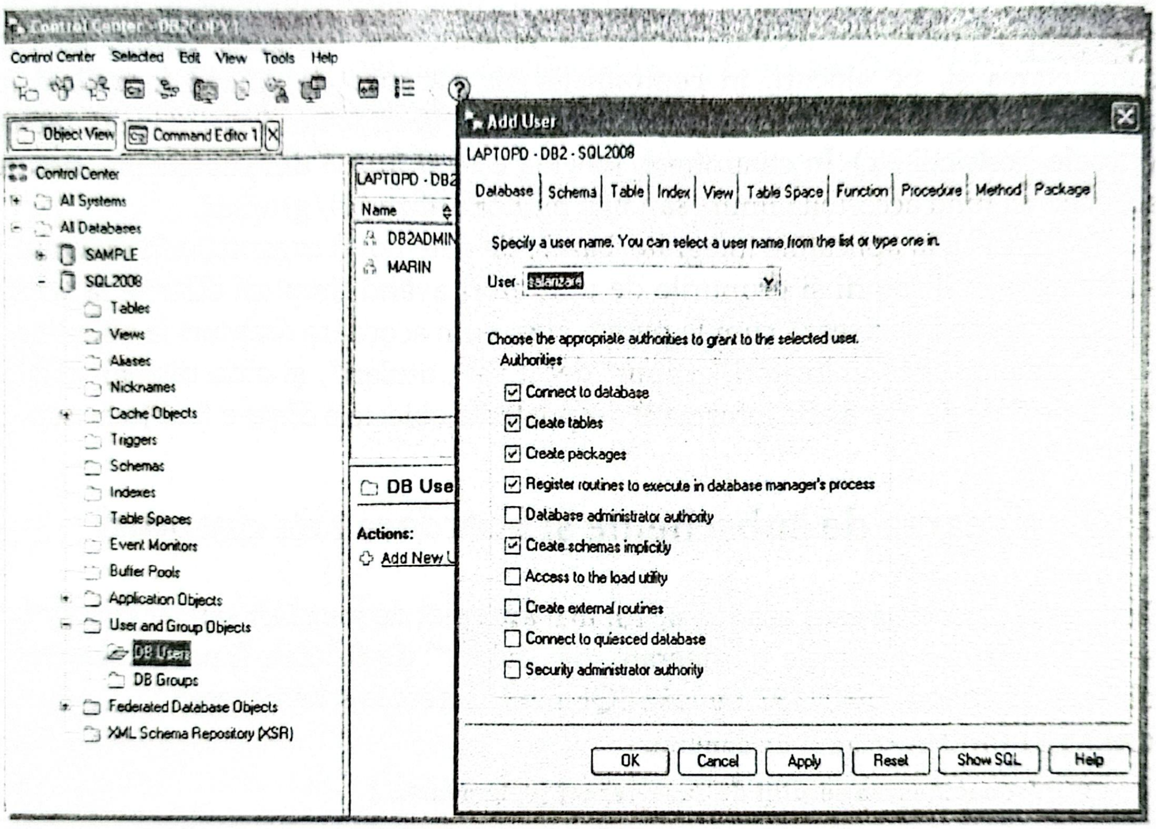


Figura 15.1. Crearea unui utilizator DB2 folosind Control Center-ul

După OK-ul de rigoare, noul utilizator apare în listă, aşa că putem trece la crearea (sub)schemei. Rămâne totuşi o întrebare: cum de nu am fost întrebaţi nimic despre parolă? Vom vedea foarte curând că acesta este un semn rău.

Comanda de creare a schemei este destul de simplă. Adăugăm şi o comandă COMMENT:

```
CREATE SCHEMA salarizare AUTHORIZATION salarizare;
COMMENT ON Schema SALARIZARE IS 'fara comentarii';
```

Crearea se putea derula şi grafic, cu acelaşi meniu arborescent de mai sus, din care alegem opţiunea *Schemas* (vezi figura 15.2). La rubrica *Authorization name* indicăm utilizatorul „titular" al schemei – *salarizare*.

După creare, încercăm să acordăm drepturi utilizatorului *salarizare* la două dintre tabelele schemei curente (schemă curentă care se numeşte *MARIN*):

GRANT ALL ON TABLE personal2 TO USER salarizare;

GRANT ALL ON TABLE sporuri TO USER salarizare;

În continuare, schimbăm subschema curentă, declarând-o pe cea nou creată - SALARIZARE:

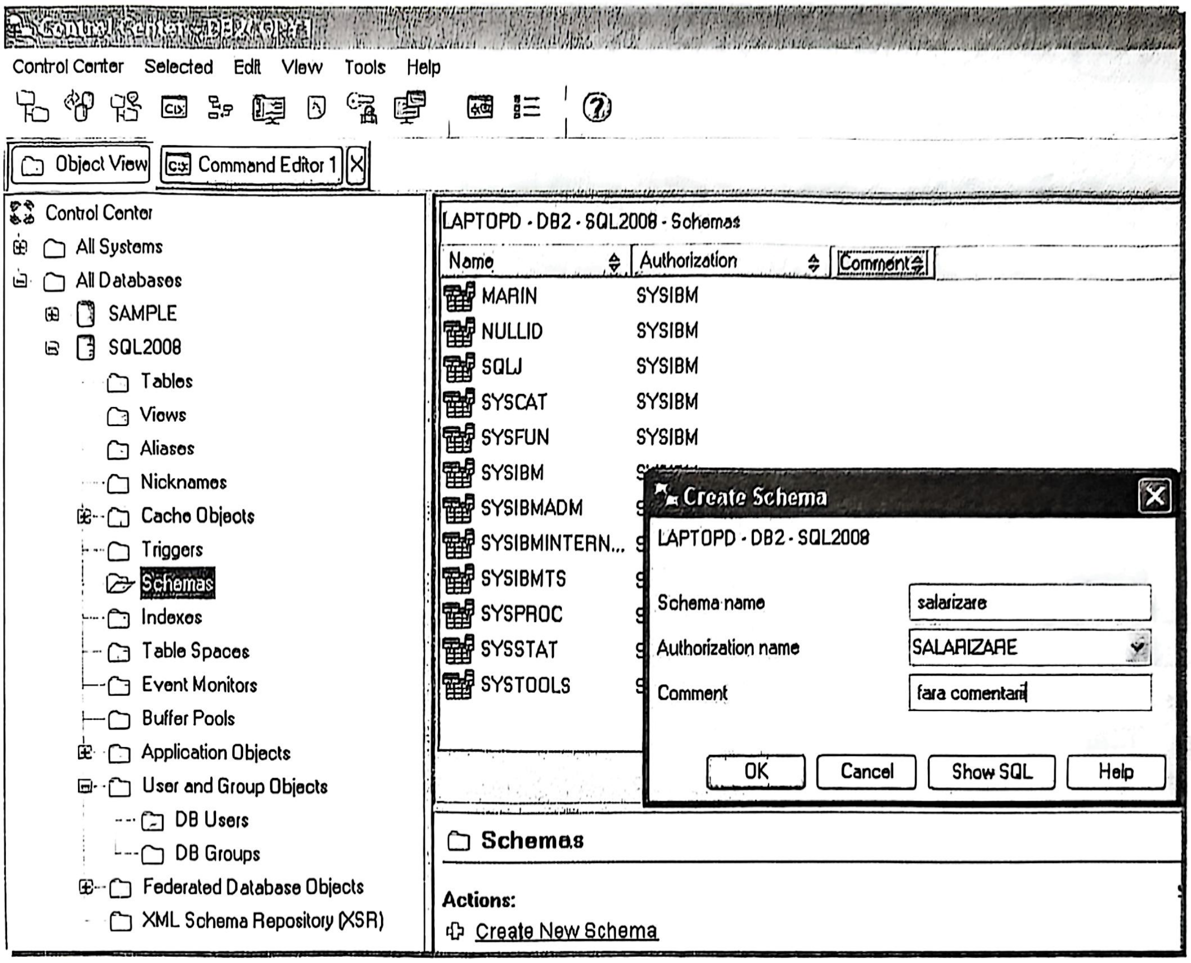


Figura 15.2. Crearea unei (sub)scheme DB2 folosind Control Center-ul

SET CURRENT_SCHEMA salarizare;

Chiar dacă subschema curentă este acum SALARIZARE, faptul că utilizatorul curent este *marin*, tabelele (sub)schemei MARIN pot fi interogare, indiferent dacă le prefixăm sau nu:

SELECT * FROM marin.personal2 sau SELECT * FROM personal2

Necazurile apar dacă încercăm să ne conectăm (la serverul care se numeşte, în cazul nostru, *SQL2008*) cu numele noului utilizator:

CONNECT TO sql2008 USER salarizare

pentru că ni se solicită parola utilizatorului (vezi figura 15.3), iar noi nu ne amintim să fi precizat aşa ceva. Indiferent ce tastăm la rubrica *Password*, vom primi un mesaj de eroare: *SQL30082N Security processing failed with reason* "24" (*"USERNAME AND/OR PASSWORD INVALID"*).

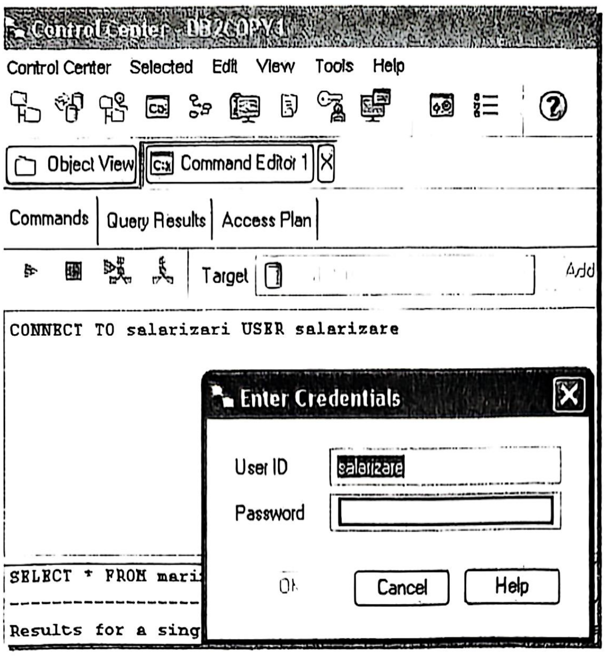


Figura 15.3. Solicitarea parolei pentru utilizatorul declarat odată cu subschema

Explicația fenomenului paranormal este că utilizatorii DB2 sunt preluați din sistemul de operare, aşa că degeaba îi declarăm noi în premieră în comenzi CREATE SCHEMA. Aşadar, trebuie să declarăm în Windows XP (acesta este sistemul de operare de pe calculatorul meu) un cont pentru utilizatorul *salarizare*, pe filiera *Control Panel → User Account*. Pentru aceasta, aveți nevoie de drepturi serioase în Windows. Contul utilizatorului nou-creat nu trebuie să fie neapărat înzestrat cu drepturi de administrator.

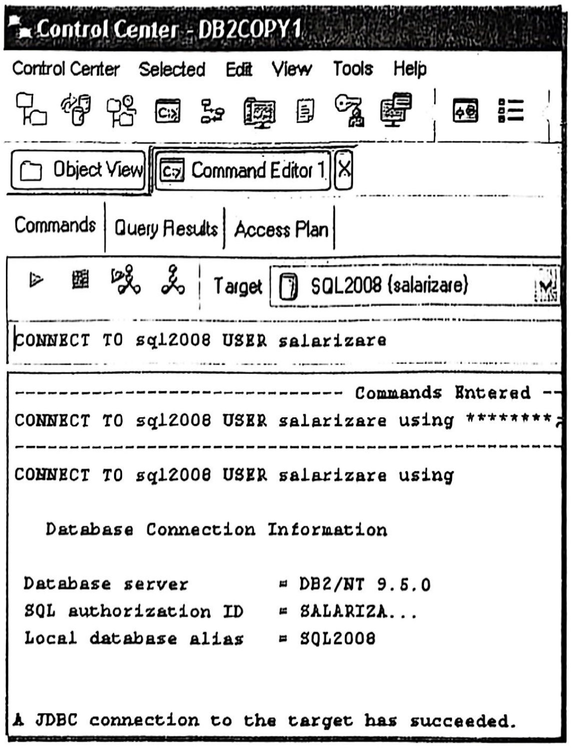


Figura 15.4. Conectarea ca utilizator *salarizare*, după crearea contului în Windows

Parola din Windows va fi furnizată acum la reconectarea cu serverul DB2 (re-execuţia comenzii CONNECT de mai sus). Dacă ţinem minte parola indicată la crearea contului în sistemul de operare, sunt şanse mari să avem un mesaj plăcut, de genul celui din figura 15.4. După conectare însă, accesul la tabelele din subschema *MARIN* este condiţionat, pe de o parte, de drepturile de acces ale utilizatorului *salarizare* şi, pe de altă parte, de prefixarea numelui tabelei cu denumirea subschemei în care se află. Astfel, interogarea:

SELECT * FROM personal2

nu funcţionează, tabela PERSONAL2 nefiind recunoscută (mesajul de eroare este: *SQL0204N "SALARIZARE.PERSONAL2" is an undefined name*). Varianta corectă a interogării este:

SELECT * FROM marin.personal2

După cum vă amintiţi, prin două comenzi GRANT am dat drepturi depline utilizatorului *salarizare* doar la două tabele din schema *MARIN*, PERSONAL2 şi SPORURI. Orice încercare de a interoga alte tabele din schema MARIN sunt sortite eşecului (vezi figura 15.5):

SELECT * FROM marin.facturi

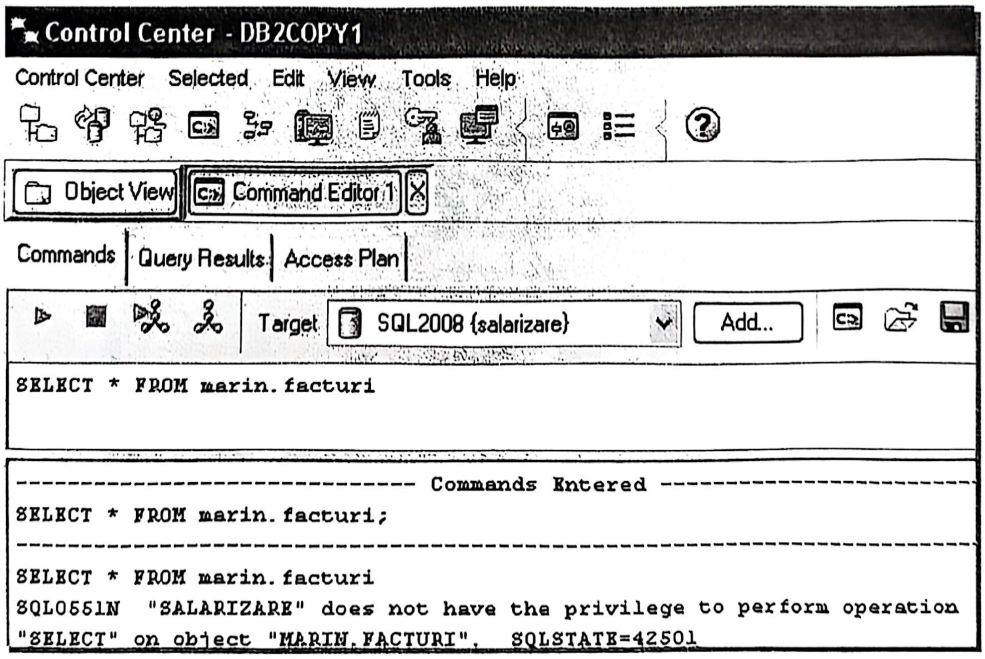


Figura 15.5. Acces interzis la o tabelă la care utilizatorul curent nu are drepturi (deloc)

15.1.2. Utilizatori şi subscheme în Oracle

În Oracle, legătura dintre schemă şi utilizator este foarte strânsă. Utilizatorii pot fi creaţi mult mai simplu decât în DB2, prin comanda CREATE USER sau interactiv. La declarare, fiecărui USER i se alocă implicit o subschemă proprie în care vor fi grupate toate obiectele pe care le va crea. Pot crea noi conturi-utilizator numai cei care au acest drept. Astfel, după conectarea ca superutilizator (SYS) sau orice alt utilizator cu drepturi corespunzătoare, putem crea contul dedicat utilizatorului

SALARIZARE şi apoi să acordăm câteva drepturi (minimale) prin secvenţa de comenzi:

```
CREATE USER salarizare IDENTIFIED BY salarizare;

GRANT CONNECT, RESOURCE, CREATE SESSION TO salarizare;
GRANT CREATE SESSION TO salarizare;
GRANT ALL ON sql2008.personal2 TO salarizare;
GRANT ALL ON sql2008.sporuri TO salarizare;
```

Iată cum arată dialogul în figura 15.6. Crearea utilizatorilor poate fi făcută şi interactiv, alegând din meniul arborescent din stânga (jos) a figurii opţiunea *Other Users*. Privilegiul (dreptul) CREATE SESSION a fost acordat de două ori utilizatorului *salarizare*, fără ca repetiţia să deranjeze.

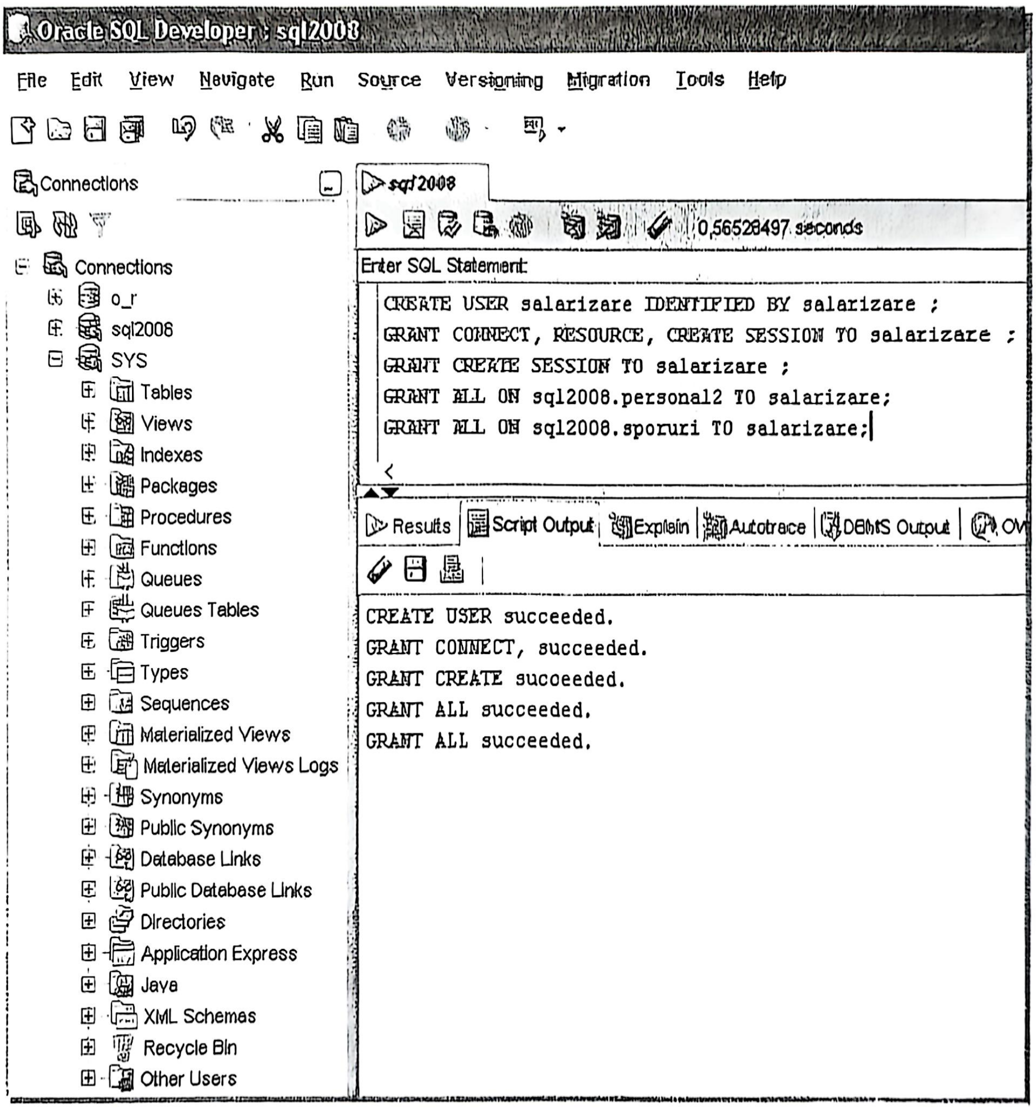


Figura 15.6. Crearea unui utilizator şi acordarea de drepturi în Oracle SQL Developer

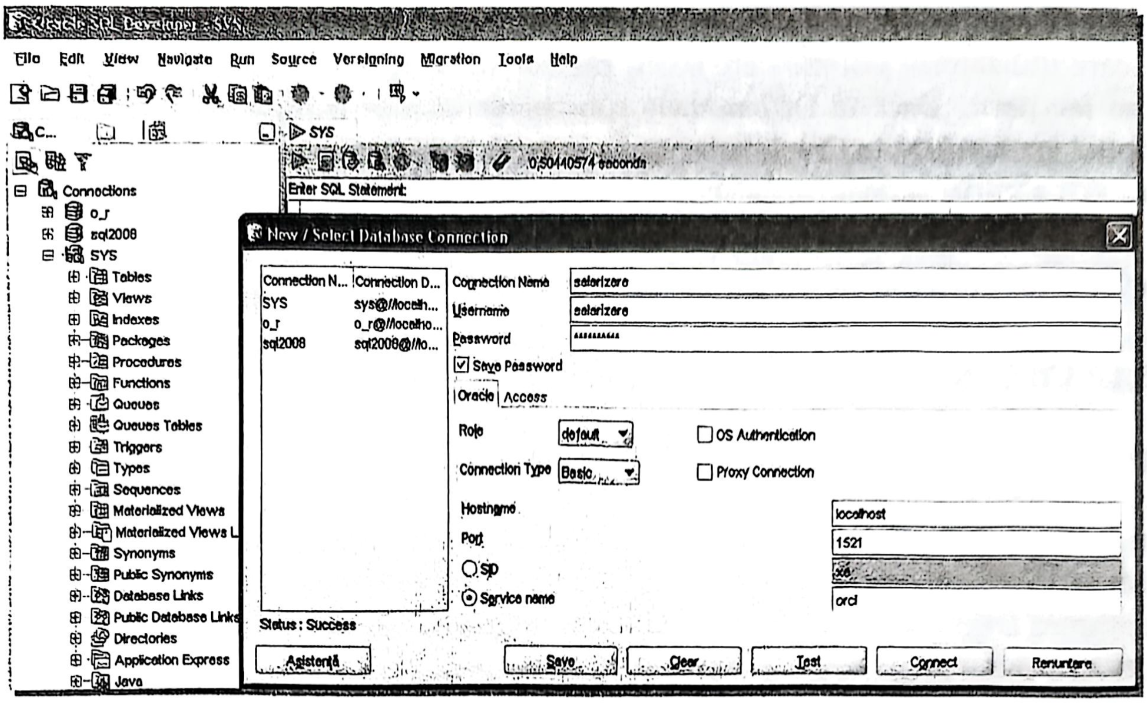


Figura 15.7. Crearea unei conexiuni pentru utilizatorul/schema salarizare

Pentru a lucra efectiv în această (sub)schemă, definim o conexiune nouă. Pentru simplitate, numele conexiunii va fi acelaşi ca numele subschemei şi utilizatorului. Cum eu lucrez cu serverul Oracle (la fel şi cu celelate trei servere BD) instalat pe un laptop pentru această carte, la rubrica *Host name* apare *localhost* iar numele serviciului (*Service name*) este cel implicit de la instalare (*orcl*). Formularul complet pentru declararea conexiunii este cel din figura 15.7. În practică, serverul BD este instalat pe o „maşină" specială, iar rubricile *Host name, Service name* etc. pot prezenta diferenţe sensibile faţă de situaţia noastră.

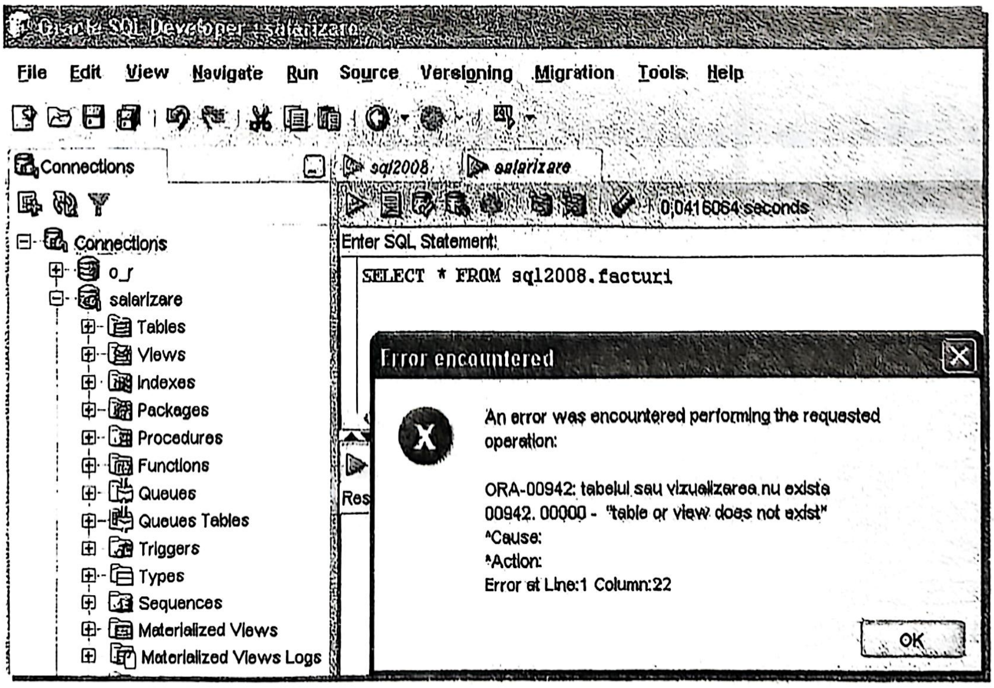


Figura 15.8. Interzicerea accesului la o tabelă neautorizată

După crearea conexiunii şi deschiderea sa, accesul se poate face doar la tabelele la care utilizatorul *salarizare* are acces, prefixând numele obiectelor cu schema din care fac parte. Dacă în DB2 numele subschemei în care am lucrat până în acest capitol era *MARIN*, în Oracle subschema se numeşte *SQL2008*.

SELECT * FROM sql2008.personal2;

Ca şi în DB2, încercarea de a consulta alte tabele decât cele două la care au fost definite drepturi este descurajată (vezi figura 15.8).

15.1.3. Utilizatori şi subscheme în PostgreSQL

Specifică PostgreSQL-ului este echivalenţa dintre comenzile CREATE USER şi CREATE ROLE. Pot crea conturi pentru noi utilizatori numai *superuser*-ii. Utilizatorul în numele căruia am lucrat până în acest paragraf se numeşte *postgres*. Vom crea noul user - *salarizare* - prin comanda CREATE USER, deşi puteam lucra şi interactiv (opţiunea *Login Roles* din meniul arborescent pgAdmin - vezi figura 15.9):

CREATE USER salarizare LOGIN PASSWORD 'salarizare';

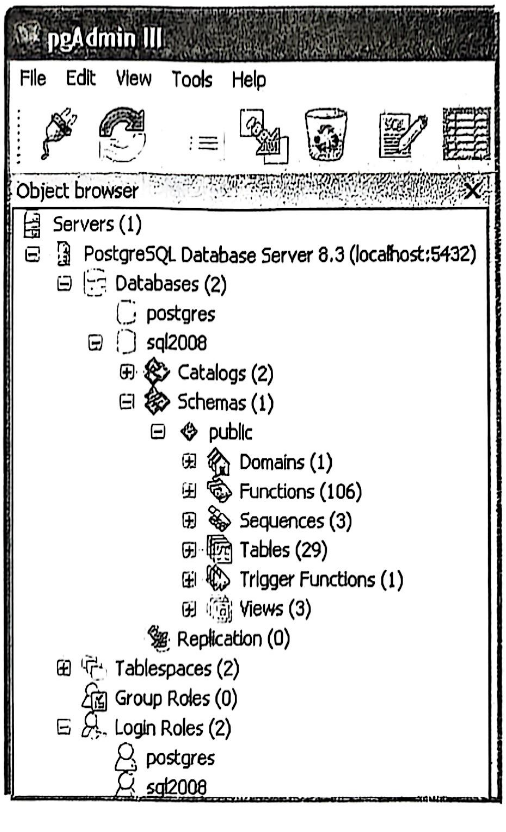


Figura 15.9. Meniul arborescent pgAdmin

Tot din figură aflăm numele schemei în care am lucrat până acum - *public* - din baza de date *sql2008*. Din paragraful 6.6 (şi figura 6.44) ştim că avem câteva funcţii PostgreSQL prin care aflăm, printre altele, numele bazei de date şi schemei curente (CURRENT_DATABASE() şi CURRENT_SCHEMA()).

Creăm în BD SQL2008, alături de *public*, subschema *salarizare*:

CREATE SCHEMA salarizare AUTHORIZATION salarizare;

Drepturile depline la cele două tabele din schema *public* trebuie precedate de acordarea dreptului la nivel de (sub)schemă:

GRANT USAGE ON SCHEMA public TO salarizare;

GRANT ALL PRIVILEGES ON public.personal2 TO salarizare;

GRANT ALL PRIVILEGES ON public.sporuri TO salarizare;

Pentru a testa modul în care funcţionează mecanismul de acordare a drepturilor, creăm o conexiune dedicată utilizatorului *salarizare* (vezi figura 15.10). Numele conexiunii este acelaşi cu al utilizatorului. Similar Oracle, la rubrica *Host* se introduce *localhost*. Numele serviciului este opţional, iar bifarea opţiunii *Connect now* asigură activarea imediată a conexiunii.

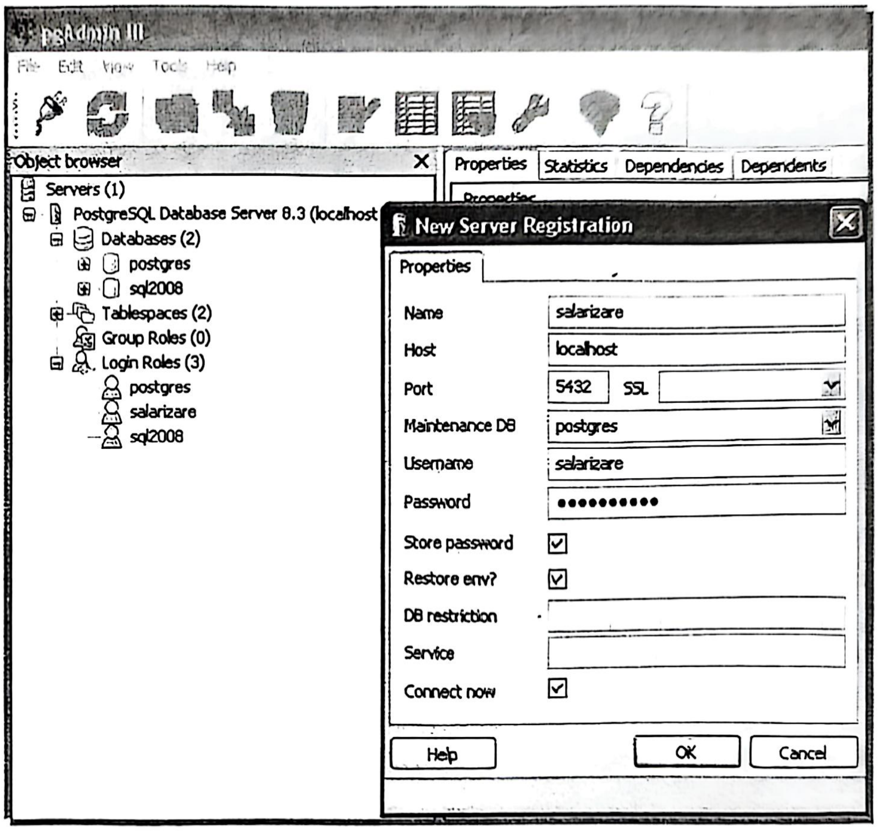


Figura 15.10. Crearea unei conexiuni (pgAdmin) pentru utilizatorul *salarizare*

După activarea conexiunii şi selectarea schemei curente (vezi figura 15.11), ştim că suntem conectaţi ca utilizator *salarizare* şi vom avea drepturile de acces şi creare/editare ale acestuia. Interogăm tabelele din subschema *public* prin oricare dintre variantele următoare:

SELECT * FROM sql2008.public.personal2;

SELECT * FROM public.personal2;

SELECT * FROM personal2;

Dacă însă încercăm să accesăm orice altă tabelă decât cele două (PERSONAL2 şi SPORURI) la care utilizatorul *salarizare* are drepturi, ne procopsim, cum era de aşteptat, cu un mesaj de eroare de genul celui din figura 15.11.

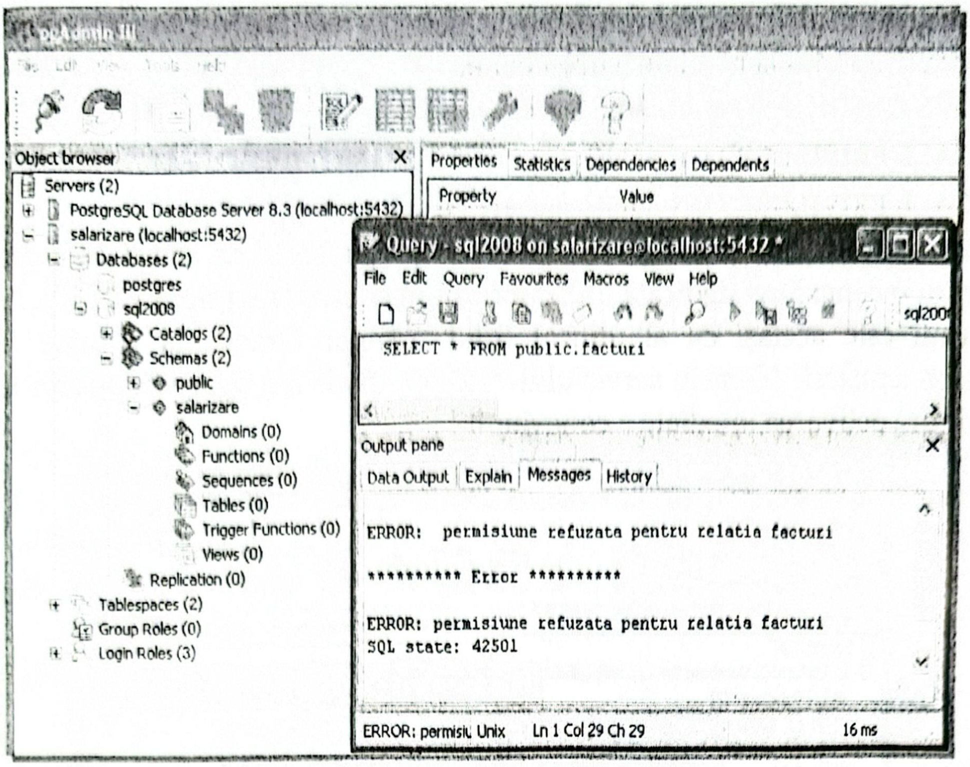


Figura 15.11. Blocarea accesului la o tabelă neautorizată în PostgreSQL

15.1.4. Utilizatori şi subscheme în SQL Server

Şi SQL Server are câteva particularităţi în materie de utilizatori, subscheme şi drepturi. Mai întâi, trebuie să ţinem cont de faptul că la definirea conexiunii folosite până acum am folosit autentificarea sistemului de operare. Aflarea bazei de date, a schemei şi a utilizatorului curente presupune folosirea funcţiilor descrise în paragraful 6.6 (vezi şi figura 15.12).

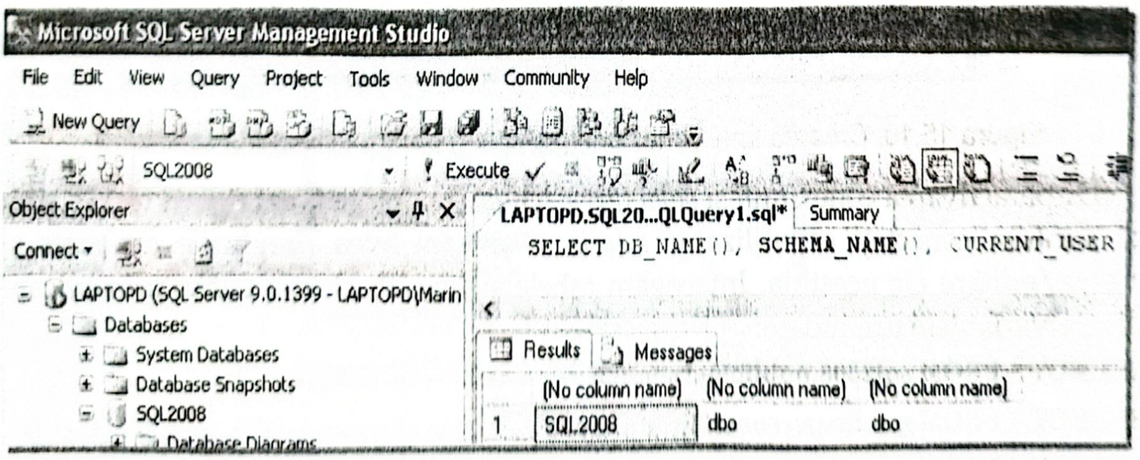


Figura 15.12. Aflarea bazei de date, a schemei şi a utilizatorului curente

O particularitate a SQL Server-ului ţine de etapele parcurse pentru crearea unui cont-utilizator. Prima etapă constă în folosirea comenzii CREATE LOGIN:

CREATE LOGIN salarizare WITH PASSWORD = 'salarizare2008';

Numai după aceasta se poate lansa CREATE USER:

CREATE USER salarizare FOR LOGIN salarizare

Noului utilizator îi vom „repartiza" o schemă cu acelaşi nume, după formatul de acum cunoscut:

CREATE SCHEMA salarizare AUTHORIZATION salarizare;

Şi în privinţa acordării drepturilor (privilegiilor), SQL Server are câteva caracteristici proprii. Vom folosi în comandă opţiunea ALL, care, în ciuda numelui, se referă doar la operaţiunile: DELETE, INSERT, REFERENCES, SELECT şi UPDATE:

GRANT ALL PRIVILEGES ON dbo.personal2 TO salarizare;

GRANT ALL PRIVILEGES ON dbo.sporuri TO salarizare;

Pentru a verifica modul de respectare a drepturilor acordate, vom crea o nouă conexiune în care vom folosi metoda de autentificare a serverului BD (vezi figura 15.13).

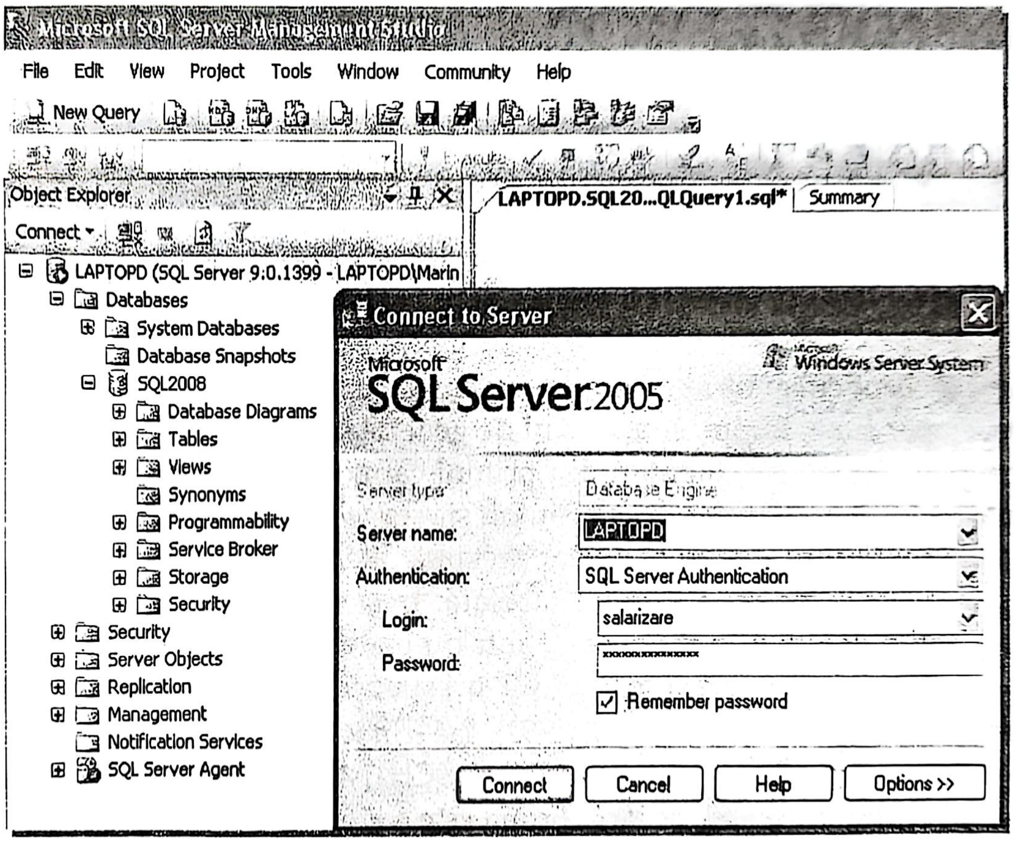


Figura 15.13. Crearea unei (alte) conexiuni SQL Server

După activarea noii conexiuni, vom vedea (figura 15.14) că nu există o schemă distinctă *salarizare*, aşa cum ne-am fi aşteptat. În schimb, meniul arborescent afişează pentru baza de date (SQL2008) numai cele două tabele la care utilizatorul *salarizare* are acces. Încercarea de a interoga orice altă tabelă decât cele două este sortită eşecului.

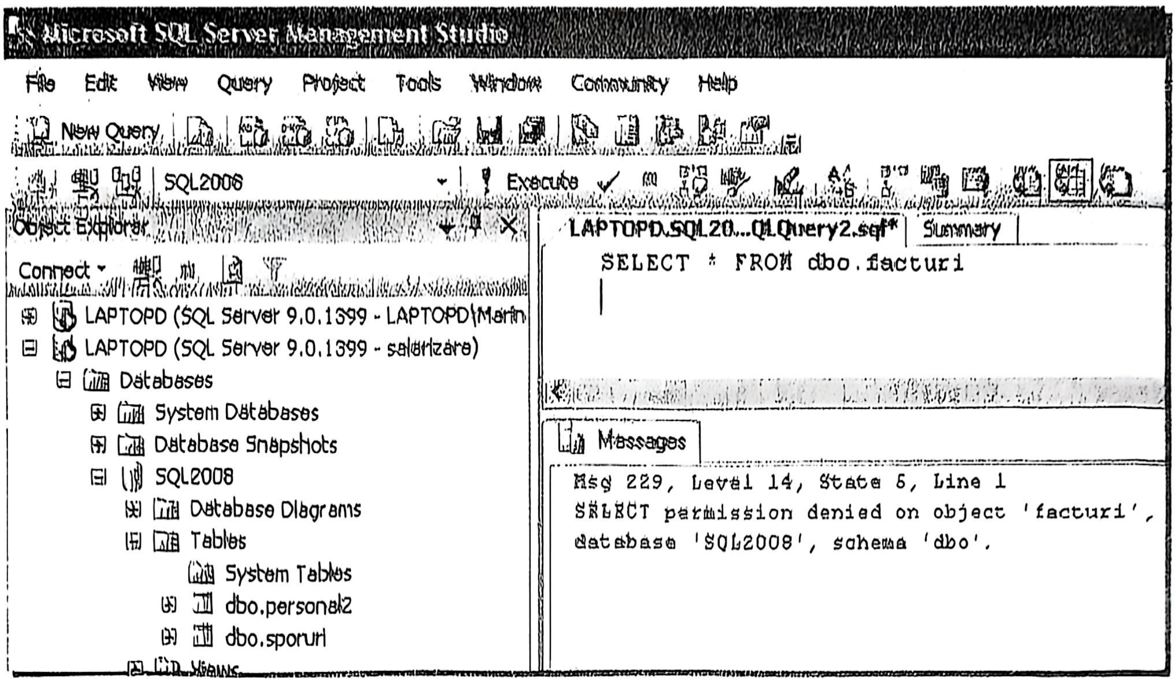


Figura 15.14. Afișarea tabelelor la care are acces utilizatorul *salarizare*

15.2. Informații despre obiectele din (sub)scheme

Programele-client cu ajutorul cărora ne putem conecta la bazele de date, cum sunt SQL Developer și pgAdmin, sunt destul de generoase în afișarea obiectelor dintr-o (sub)schemă a bazei. Ne amintim din primul capitol că întreaga structura a unei BD se păstrează în *dicționarul de date*, numit și *catalog-sistem* sau *container de date* etc. Structura și conținutul acestui dicționar prezintă diferențe sensibile de la un produs la altul, însă organizarea sa este tot tabelară. Deși vorbim despre un *dicționar de date*, în practică acesta este fragmentat într-o puzderie de tabele și tabele virtuale care, firește, pot fi interogate prin SELECT-uri și uneori chiar modificate.

Standardele SQL (începând cu SQL-92)[1] fac trimitere la două (sub)scheme speciale din orice bază de date, INFORMATION_SCHEMA și DEFINITION_ SCHEMA ([Kriegel & Trukhnov 2008], p. 495). Prima este compusă din tabele virtuale (*views*) ce furnizează informații despre obiectele persistente ale bazei, în timp ce a doua este alcătuită din tabele „obișnuite", ce conțin informații despre modul de organizare al primeia. Dicționarul de date este deci denumit în SQL *catalog*, catalog ce reprezintă o colecție de (sub)scheme printre care se regăsesc și cele două menționate.

Fiecare tabelă virtuală sau tabelă din catalog este specializată în anumite categorii de informații (vezi [Kriegel & Trukhnov 2008], pp. 496-500). Iată câteva exemple:

TABLES - informații generale despre tabele;

1. Nu uitați că, în afara primelor două părți generale, există o parte a standardului SQL numită *SQL/Schemata*.

COLUMNS - informaţii despre atribute;
TABLE_CONSTRAINTS - informaţii despre restricţiile definite în tabele;
CHECK_CONSTRAINTS - informaţii numai despre restricţiile definite prin clauza CHECK (reguli de validare la nivel de atribut şi înregistrare);
REFERENTIAL_CONSTRAINTS - informaţii numai despre restricţiile referenţiale;
SCHEMATA - subschemele dintr-o bază de date;
VIEWS - informaţii despre tabelele virtuale.

Din păcate, denumirile lor sunt cât de poate de eterogene de la un SGBD la altul, după cum vom vedea în continuare.

15.2.1. Obiecte din schema unei baze de date în PostgreSQL

Începem cu descrierea câtorva informaţii din dicţionarul de date PostgreSQL. Punctul de plecare în aflarea pe cale „interogaţională" a informaţiilor din schema unei baze de date PostgreSQL este tabela-sistem PG_TABLES:

SELECT * FROM pg_tables

schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean
information_schema	sql_features	postgres		f	f	f
information_schema	sql_implementation_info	postgres		f	f	f
pg_catalog	pg_statistic	postgres		t	f	f
information_schema	sql_languages	postgres		f	f	f
information_schema	sql_packages	postgres		f	f	f
information_schema	sql_parts	postgres		f	f	f
information_schema	sql_sizing	postgres		f	f	f
pg_catalog	pg_authid	postgres	pg_global	t	f	t
information_schema	sql_sizing_profiles	postgres		f	f	f
pg_catalog	pg_database	postgres	pg_global	t	f	t
pg_catalog	pg_tablespace	postgres	pg_global	t	f	f
pg_catalog	pg_pltemplate	postgres	pg_global	t	f	f
pg_catalog	pg_shdepend	postgres	pg_global	t	f	f
pg_catalog	pg_shdescription	postgres	pg_global	t	f	f
pg_catalog	pg_ts_config	postgres		t	f	f
pg_catalog	pg_ts_config_map	postgres		t	f	f
pg_catalog	pg_ts_dict	postgres		t	f	f
pg_catalog	pg_ts_parser	postgres		t	f	f
pg_catalog	pg_ts_template	postgres		t	f	f
pg_catalog	pg_auth_members	postgres	pg_global	t	f	t
pg_catalog	pg_type	postgres		t	f	f
pg_catalog	pg_attribute	postgres		t	f	f
pg_catalog	pg_proc	postgres		t	f	f

Figura 15.15. Tabela centrală a dicţionarului de date în PostgreSQL

O parte din conţinutul său este afişată în figura 15.15. Informaţiile sunt dispuse în trei subscheme, INFORMATION_SCHEMA, PG_CATALOG şi PUBLIC (care nu a încăput în figură).

Tabelele-dicţionar din (sub)schema INFORMATION_SCHEMA sunt de interes general, cum ar fi SQL_PACKAGES care informează despre modul de implemen-

tare în versiunea curentă de PostgreSQL a componentelor din standardul SQL (vezi figura 15.16):

SELECT * FROM information_schema.sql_packages

feature_id character vai	feature_name character varying	is_supported character vai	is_verified_by character vary	comments character varying
PKG000	Core	NO		
PKG001	Enhanced datetime facilities	YES		
PKG002	Enhanced integrity management	NO		
PKG003	OLAP facilities	NO		
PKG004	PSM	NO		PL/pgSQL is similar.
PKG005	CLI	NO		ODBC is similar.
PKG006	Basic object support	NO		
PKG007	Enhanced object support	NO		
PKG008	Active database	NO		
PKG010	OLAP	NO		NO

Figura 15.16. Aflarea modului de implementare în PostgreSQL a componentelor din standardul SQL

sau SQL_FEATURES care furnizează răspunsul la întrebarea *care dintre opţiunile standardului SQL sunt implementate şi care nu* etc. În subschema PUBLIC se află tabelele create de utilizator (adică de noi) până la momentul curent. Dacă le-aţi uitat cumva, le putem obţine foarte simplu (vezi figura 15.17):

SELECT * FROM **pg_tables WHERE schemaname='public' ORDER BY tablename**

schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean
public	clienti	postgres		t	f	t
public	coduri_postale	postgres		t	f	t
public	curse	postgres		f	f	f
public	curse_statii	postgres		f	f	f
public	distante	postgres		t	f	f
public	doctori	postgres		t	f	t
public	facturi	postgres		t	f	t
public	facturi_2006	postgres		f	f	f
public	facturi_2007	postgres		f	f	f
public	garzi	postgres		t	f	t
public	incasari	postgres		t	f	t
public	incasfact	postgres		t	f	t
public	judete	postgres		t	f	t
public	liniifact	postgres		t	f	t
public	liniifact_2007	postgres		f	f	f
public	pacienti	postgres		t	f	t

Figura 15.17. O parte dintre tabelele create de utilizator (adică de noi)

Detaliile despre celelalte ingrediente ale structurii bazei - atribute, restricţii, proceduri stocate, declanşatoare etc. se găsesc în tabele specializate ale dicţionarului, tabele aflate în schema PG_CATALOG. Conţinutul acestora este, de cele mai multe ori derutant. Spre exemplu, dacă dorim să aflăm ceva mai multe despre atributele din tabelele create în capitolele anterioare, vom începe prin a scotoci în PG_ATTRIBUTE:

SELECT * FROM pg_catalog.pg_attribute

După cum se poate zări în figura 15.18, conţinutul acestei tabele este devastator, aici zăcând, de-a valma, atributele „noastre" cu atribute din tabele-sistem, greu de contabilizat şi comprehensibile (sesizaţi, vă rog, termenul elevat!).

	attrelid oid	attname name	atttypid oid	attstattarget integer	attlen smallint	attnum smallint	attndims integer	attcacheoff integer	atttypm integer
2237	26117	tableoid	26	0	4	-7	0	-1	-1
2238	26119	chunk_id	26	-1	4	1	0	-1	-1
2239	26119	chunk_seq	23	-1	4	2	0	-1	-1
2240	17785	valtotala	1700	-1	-1	5	0	-1	655366
2241	17785	tva	1700	-1	-1	6	0	-1	655366
2242	17785	valincasata	1700	-1	-1	7	0	-1	655366
2243	17785	reduceri	1700	-1	-1	8	0	-1	589830
2244	17785	penalizari	1700	-1	-1	9	0	-1	589830
2245	17797	tvainie	1700	-1	-1	6	0	-1	589830
2246	26178	jud	1042	-1	-1	1	0	-1	6
2247	26178	judet	1043	-1	-1	2	0	-1	29
2248	26178	regiune	1043	-1	-1	3	0	-1	19

Figura 15.18. O (mică) parte din tabela (virtuală)-dicţionar PG_CATALOG.PG_ATTRIBUTE

Filtrarea atributelor după numele tabelei din care fac parte e ceva mai laborioasă. Ne bazăm pe coloana *Attrelid* (prima din figura 15.18) care conţine identificatorul tabelei. Indicatorul şi numele tabelei sunt conţinute în tabela-dicţionar PG_CLASS, aşa că interogarea va avea forma:

SELECT * FROM pg_attribute

WHERE attrelid = (SELECT oid FROM pg_class WHERE relname='facturi')

Însă tot n-am scăpat de atributele-sistem ale tabelei (vezi figura 15.19). O soluţie ar exista, totuşi. Numărul fiecărui atribut din tabelă – *Attnum* – este pozitiv, dacă se numără printre cele definite de utilizator, şi negativ, dacă este unul sistem:

	attrelid oid	attname name	atttypid oid	attstattarget integer	attlen smallint	attnum smallint	attndims integer	attcacheoff integer	atttypmod integer	attby boole
1	17785	tableoid	26	0	4	-7	0	-1	-1	t
2	17785	cmax	29	0	4	-6	0	-1	-1	t
3	17785	xmax	28	0	4	-5	0	-1	-1	t
4	17785	cmin	29	0	4	-4	0	-1	-1	t
5	17785	xmin	28	0	4	-3	0	-1	-1	t
6	17785	ctid	27	0	6	-1	0	-1	-1	f
7	17785	nrfact	23	-1	4	1	0	-1	-1	t
8	17785	datafact	1082	-1	4	2	0	-1	-1	t
9	17785	codcl	21	-1	2	3	0	-1	-1	t
10	17785	obs	1043	-1	-1	4	0	-1	54	f
11	17785	valtotala	1700	-1	-1	5	0	-1	655366	f
12	17785	tva	1700	-1	-1	6	0	-1	655366	f
13	17785	valincasata	1700	-1	-1	7	0	-1	655366	f
14	17785	reduceri	1700	-1	-1	8	0	-1	589830	f
15	17785	penalizari	1700	-1	-1	9	0	-1	589830	f

Figura 15.19. Atributele tabelei FACTURI

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'facturi')
        AND attnum > 0
ORDER BY attnum
```

Pentru a afişa numele, tipul şi valoarea implicită a fiecărui atribut din tabela FACTURI (figura 15.20), este necesar să recurgem şi la tabelele virtuale-dicţionar PG_TYPE şi PG_ATTRDEF:

```
SELECT attname AS NumeAtribut, typname AS tip,
        (SELECT adsrc FROM pg_attrdef WHERE (adrelid, adnum) = (attrelid, attnum))
                AS val_implicita
FROM pg_attribute INNER JOIN pg_type ON pg_attribute.atttypid = pg_type.typelem
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'facturi') AND attnum > 0
ORDER BY attnum
```

numeatribut name	tip name	val_implicita text
nrfact	_int4	
datafact	_date	('now'::text)::date
codcl	int2vector	
codcl	_int2	
obs	_varchar	
valtotala	_numeric	0
tva	_numeric	0
valincasata	_numeric	0
reduceri	_numeric	0
penalizari	_numeric	0

Figura 15.20. Informaţii despre atributele tabelei FACTURI aflate din *view*-urile dicţionar *pg_attribute*, *pg_type* şi *pg_attrdef*

Mult mai simplu este, însă, să folosim tabela virtuală *COLUMNS* din (sub)schema INFORMATION_SCHEMA:

```
SELECT column_name, data_type,
        CASE
        WHEN data_type LIKE 'char%' THEN character_maximum_length || ''
        WHEN data_type = 'numeric' THEN numeric_precision
        || '(' || numeric_scale || ')' END AS Lungime,
        is_nullable
FROM information_schema.columns
WHERE table_catalog = 'sql2008' AND table_schema = 'public' AND table_name = 'facturi'
ORDER BY ordinal_position
```

Cu această versiune nu numai că suntem mai aproape de sintaxa standardului, dar şi modul în care se prezintă informaţiile este parcă mai arătos (vezi figura 15.21).

column_name character varying	data_type character var	lungime text	is_nullable character va
nrfact	integer		NO
datafact	date		YES
codcl	smallint		YES
obs	character varyir	50	YES
valtotala	numeric	10(2)	YES
tva	numeric	10(2)	YES
valincasata	numeric	10(2)	YES
reduceri	numeric	9(2)	YES
penalizari	numeric	9(2)	YES

Figura 15.21. Informaţii despre atribute aflate din view-ul *information_schema.columns*

Afişarea regulilor de validare (clauzele CHECK) dintr-o tabelă (vezi figura 15.22) reclamă folosirea a două noi tabele virtuale dicţionar: CHECK_CONSTRAINTS şi CONSTRAINT_COLUMN_USAGE.

```
SELECT DISTINCT
        (SELECT ordinal_position FROM information_schema.columns
                WHERE table_catalog = 'sql2008' AND table_schema = 'public' AND
                        table_name = 'facturi' AND column_name = ccu.column_name)
                AS NrCrt,
        column_name,
        (SELECT check_clause FROM information_schema.check_constraints
        WHERE constraint_name = ccu.constraint_name) AS clauza_CHECK
FROM information_schema.constraint_column_usage ccu
WHERE table_catalog = 'sql2008' AND table_schema = 'public' AND table_name = 'facturi'
ORDER BY 1
```

nrcrt integer	column_name character vary	clauza_check character varying
1	nrfact	
2	datafact	(((datafact >= to_date('2007-01-01'::text, 'YYYY-MM-DD'::text)) AND (datafact <= '2015-12-31'::date)))

Figura 15.22. Restricţiile de tip CHECK definite în tabela FACTURI

Cea mai simplă modalitate de a obţine toate restricţiile definite într-o tabelă presupune interogarea tabelei virtuale TABLE_CONSTRAINTS. Iată restricţiile tabelei LINIIFACT (vezi figura 15.23):

constraint_catalog character varying	constraint_schema character varying	constraint_name character varying	table_catalog character vary	table_schema character vary	table_name character var	constraint_type character varyi	is_deferrable character var	initially_deferred character varying
sql2008	public	2200_17797_1_not_null	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	2200_17797_2_not_null	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	2200_17797_3_not_null	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	2200_17797_4_not_null	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	2200_17797_5_not_null	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	ck_linio	sql2008	public	liniifact	CHECK	NO	NO
sql2008	public	fk_liniifact_facturi	sql2008	public	liniifact	FOREIGN KEY	NO	NO
sql2008	public	fk_liniifact_produse	sql2008	public	liniifact	FOREIGN KEY	NO	NO
sql2008	public	pk_liniifact	sql2008	public	liniifact	PRIMARY KEY	NO	NO
sql2008	public	un_liniifact	sql2008	public	liniifact	UNIQUE	NO	NO

Figura 15.23. Restricţii definite în tabela LINIIFACT

```
SELECT *
FROM information_schema.table_constraints
WHERE table_name = 'liniifact'
```

O parte dintre cheile primare (PRIMARY KEY) şi alternative (UNIQUE) sunt compuse, aşa că, pentru a afla expresiile (atributele din componenţa cheilor) apelăm la joncţiunea externă repetată a tabelei virtuale TABLE_CONSTRAINTS cu KEY_COLUMN_USAGE:

```
SELECT tc.constraint_name, '(' || kcu1.column_name ||
        CASE WHEN kcu2.column_name IS NULL THEN''
        ELSE ','||kcu2.column_name END
                || CASE WHEN kcu3.column_name IS NULL THEN''
        ELSE ','||kcu3.column_name END
                || CASE WHEN kcu4.column_name IS NULL THEN''
        ELSE ','||kcu4.column_name END
                ||')' AS expresie_restrictie
FROM information_schema.table_constraints tc
        INNER JOIN information_schema.key_column_usage kcu1
                ON tc.constraint_name = kcu1.constraint_name
                        AND kcu1.ordinal_position = 1
        LEFT OUTER JOIN information_schema.key_column_usage kcu2
                ON tc.constraint_name = kcu2.constraint_name
                        AND kcu2.ordinal_position = 2
        LEFT OUTER JOIN information_schema.key_column_usage kcu3
                ON tc.constraint_name = kcu3.constraint_name
                        AND kcu3.ordinal_position = 3
        LEFT OUTER JOIN information_schema.key_column_usage kcu4
                ON tc.constraint_name = kcu4.constraint_name
                        AND kcu4.ordinal_position = 4
WHERE tc.table_name = 'liniifact'
```

Interogarea de mai sus se bazează pe faptul că în tabelele bazei de date VÂNZĂRI orice cheie primară, alternativă sau străină nu are mai mult de patru atribute (de fapt, numărul maxim este trei, dar parcă ar fi prea simplu). Rezultatul este cel din figura 15.24.

constraint_name character varying	expresie_restrictie text
fk_liniifact_produse	(codpr)
fk_liniifact_facturi	(nrfact)
un_liniifact	(nrfact,codpr)
pk_liniifact	(nrfact,linie)

Figura 15.24. Expresiile cheii primare, alternative şi ale celor străine din tabela LINIIFACT

Expresia cheilor străine este incompletă, deoarece lipsesc numele tabelei-părinte şi atributele din cheia primară/alternativă a tabelei-părinte. Aşa că intră în scenă o altă tabelă virtuală din INFORMATION_SCHEMA, şi anume REFERENTIAL_CONSTRAINTS (vezi figura 15.25):

```
SELECT * FROM information_schema.referential_constraints
```

constrair characte	constrai charact	constraint_name character varying	unique_ characte	uniqu chara	unique_const character var	match_option character varying	update_rule character var	delete_rule character va
sql2008	public	fk_personal2	sql2008	public	pk_personal2	NONE	NO ACTION	NO ACTION
sql2008	public	fk_sporuri_pers2	sql2008	public	pk_personal2	NONE	NO ACTION	NO ACTION
sql2008	public	fk_coduri_post_jud	sql2008	public	pk_judete	NONE	CASCADE	RESTRICT
sql2008	public	fk_persoane_cp	sql2008	public	pk_coduri_post	NONE	CASCADE	RESTRICT
sql2008	public	fk_persclienti_clienti	sql2008	public	pk_clienti	NONE	CASCADE	RESTRICT
sql2008	public	fk_persclienti_persoane	sql2008	public	pk_persoane	NONE	CASCADE	RESTRICT
sql2008	public	garzi_iddoctor_fkey	sql2008	public	pk_doctori	NONE	CASCADE	NO ACTION
sql2008	public	fk_internari_pacienti	sql2008	public	pk_pacienti	NONE	CASCADE	NO ACTION
sql2008	public	fk_clienti_cp	sql2008	public	pk_coduri_post	NONE	CASCADE	RESTRICT
sql2008	public	fk_incasfact_facturi	sql2008	public	pk_facturi	NONE	CASCADE	RESTRICT
sql2008	public	fk_incasfact_incasari	sql2008	public	pk_incasari	NONE	CASCADE	RESTRICT
sql2008	public	fk_liniifact_produse	sql2008	public	pk_produse	NONE	CASCADE	RESTRICT
sql2008	public	fk_liniifact_facturi	sql2008	public	pk_facturi	NONE	CASCADE	RESTRICT
sql2008	public	fk_facturi_clienti	sql2008	public	pk_clienti	NONE	CASCADE	NO ACTION

Figura 15.25. Conţinutul view-ului *information_schema.referential_constraints*

Deşi în baza de date toate cheile străine sunt simple, formulăm o soluţie care are funcţiona chiar şi dacă am avea chei străine alcătuite din patru atribute:

```
SELECT tc.table_name, constraint_type || ' (' || kcu1.column_name ||
        CASE WHEN kcu2.column_name IS NULL THEN '' ELSE ','||kcu2.column_name
            END ||
        CASE WHEN kcu3.column_name IS NULL THEN '' ELSE ','||kcu3.column_name
            END ||
        CASE WHEN kcu4.column_name IS NULL THEN '' ELSE ','||kcu4.column_name
            END ||') REFERENCES ' || (
            SELECT tc2.table_name || ' (' || kcu11.column_name ||
                    CASE WHEN kcu12.column_name IS NULL THEN ''
                            ELSE ','||kcu12.column_name END ||
                    CASE WHEN kcu13.column_name IS NULL THEN ''
                            ELSE ','||kcu13.column_name END ||
                    CASE WHEN kcu14.column_name IS NULL THEN ''
                            ELSE ','||kcu14.column_name END ||')'
            FROM information_schema.table_constraints tc2
                    INNER JOIN information_schema.key_column_usage kcu11
                            ON tc2.constraint_name = kcu11.constraint_name
                                    AND kcu11.ordinal_position =1
                    LEFT OUTER JOIN information_schema.key_column_usage kcu12
                            ON tc2.constraint_name = kcu12.constraint_name
```

```
                AND kcu12.ordinal_position = 2
        LEFT OUTER JOIN information_schema.key_column_usage kcu13
            ON tc2.constraint_name = kcu13.constraint_name
                AND kcu13.ordinal_position = 3
        LEFT OUTER JOIN information_schema.key_column_usage kcu14
            ON tc2.constraint_name = kcu14.constraint_name
                AND kcu14.ordinal_position = 4
    WHERE tc2.constraint_name = rc.unique_constraint_name
        ) || CASE update_rule WHEN 'NO ACTION' THEN ''
        ELSE ' ON UPDATE ' || update_rule END ||
        CASE delete_rule WHEN 'NO ACTION' THEN ''
        ELSE ' ON DELETE '|| delete_rule END
            AS "Expresie_completa_FOREIGN_KEY"
FROM (information_schema.table_constraints tc
    INNER JOIN information_schema.key_column_usage kcu1
        ON tc.constraint_name = kcu1.constraint_name
            AND kcu1.ordinal_position = 1
    LEFT OUTER JOIN information_schema.key_column_usage kcu2
        ON tc.constraint_name = kcu2.constraint_name
            AND kcu2.ordinal_position = 2
    LEFT OUTER JOIN information_schema.key_column_usage kcu3
        ON tc.constraint_name = kcu3.constraint_name
            AND kcu3.ordinal_position = 3
    LEFT OUTER JOIN information_schema.key_column_usage kcu4
        ON tc.constraint_name = kcu4.constraint_name
            AND kcu4.ordinal_position = 4)
    INNER JOIN information_schema.referential_constraints rc
        ON tc.constraint_name = rc.constraint_name
WHERE constraint_type = 'FOREIGN KEY'
ORDER BY tc.table_name
```

Sunt extrase toate cheile străine din schema *public* (vezi figura 15.26). Ca o confirmare a corectitudinii interogării (nu că s-ar fi îndoit careva!), pe linia 12 apare şi restricţia referenţială în care cheia străină şi cheia primară corespondentă se află în aceeaşi tabelă - PERSONAL2.

table_name character var	Expresie_completa_FOREIGN_KEY text
clienti	FOREIGN KEY (codpost) REFERENCES coduri_postale (codpost) ON UPDATE CASCADE ON DELETE RESTRICT
coduri_postale	FOREIGN KEY (jud) REFERENCES judete (jud) ON UPDATE CASCADE ON DELETE RESTRICT
facturi	FOREIGN KEY (codcl) REFERENCES clienti (codcl) ON UPDATE CASCADE
garzi	FOREIGN KEY (iddoctor) REFERENCES doctori (iddoctor) ON UPDATE CASCADE
incasfact	FOREIGN KEY (codinc) REFERENCES incasari (codinc) ON UPDATE CASCADE ON DELETE RESTRICT
incasfact	FOREIGN KEY (nrfact) REFERENCES facturi (nrfact) ON UPDATE CASCADE ON DELETE RESTRICT
liniifact	FOREIGN KEY (codpr) REFERENCES produse (codpr) ON UPDATE CASCADE ON DELETE RESTRICT
liniifact	FOREIGN KEY (nrfact) REFERENCES facturi (nrfact) ON UPDATE CASCADE ON DELETE RESTRICT
persclienti	FOREIGN KEY (cnp) REFERENCES persoane (cnp) ON UPDATE CASCADE ON DELETE RESTRICT
persclienti	FOREIGN KEY (codcl) REFERENCES clienti (codcl) ON UPDATE CASCADE ON DELETE RESTRICT
persoane	FOREIGN KEY (codpost) REFERENCES coduri_postale (codpost) ON UPDATE CASCADE ON DELETE RESTRICT
personal2	FOREIGN KEY (marcasef) REFERENCES personal2 (marca)
sporuri	FOREIGN KEY (marca) REFERENCES personal2 (marca)
triaj	FOREIGN KEY (idpacient) REFERENCES pacienti (idpacient) ON UPDATE CASCADE

Figura 15.26. Expresiile complete ale cheilor străine

15.2.2. Obiecte din schema unei baze de date în Oracle

Modalităţile sub care se prezintă dicţionarul de date în Oracle au fost tratate ceva mai pe larg în lucrarea [Fotache *et al.* 2003][2]. Toate tabelele şi tabelele virtuale din dicţionar aparţin (super)utilizatorului SYS, iar pentru simplificarea accesului, fiecare are un sinonim. De exemplu, în loc de SYS.DBA_TABLES se poate folosi sinonimul DBA_TABLES. Dacă numele view-urilor dicţionar este prefixat de USER, atunci este vorba despre obiecte ce aparţin utilizatorului curent; ALL desemnează, în plus, şi obiectele la care utilizatorul curent are acces, dar proprietarul este un alt utilizator, iar prefixul DBA (Data Base Administrator) se referă la toate obiectele bazei, indiferent de „proprietar". Tabela principală a catalogului este numită *DICTIONARY* şi are doar două coloane, TABLE_NAME şi COMMENTS (vezi figura 15.27). Aflarea tuturor tabelelor virtuale ce oferă informaţii despre obiectele din schemă este posibilă prin interogarea:

SELECT * FROM dictionary

Este posibil să aflăm orice informaţie despre schema bazei de date, atât despre obiecte de tipul celor create până în prezent (tabele, tabele virtuale, restricţii, indecşi), cât şi de genul celor create de aici încolo (funcţii, proceduri, pachete, declanşatoare, tipuri etc.). Totul e să ştim (sau să nimerim) tabela virtuală corespunzătoare din dicţionar. Pentru non-administratori, inventarierea obiectelor din schema proprie (tabele, tabele virtuale, sinonime, proceduri, funcţii, pachete, secvenţe etc.) este posibilă folosind tabela vituală-dicţionar USER_OBJECTS:

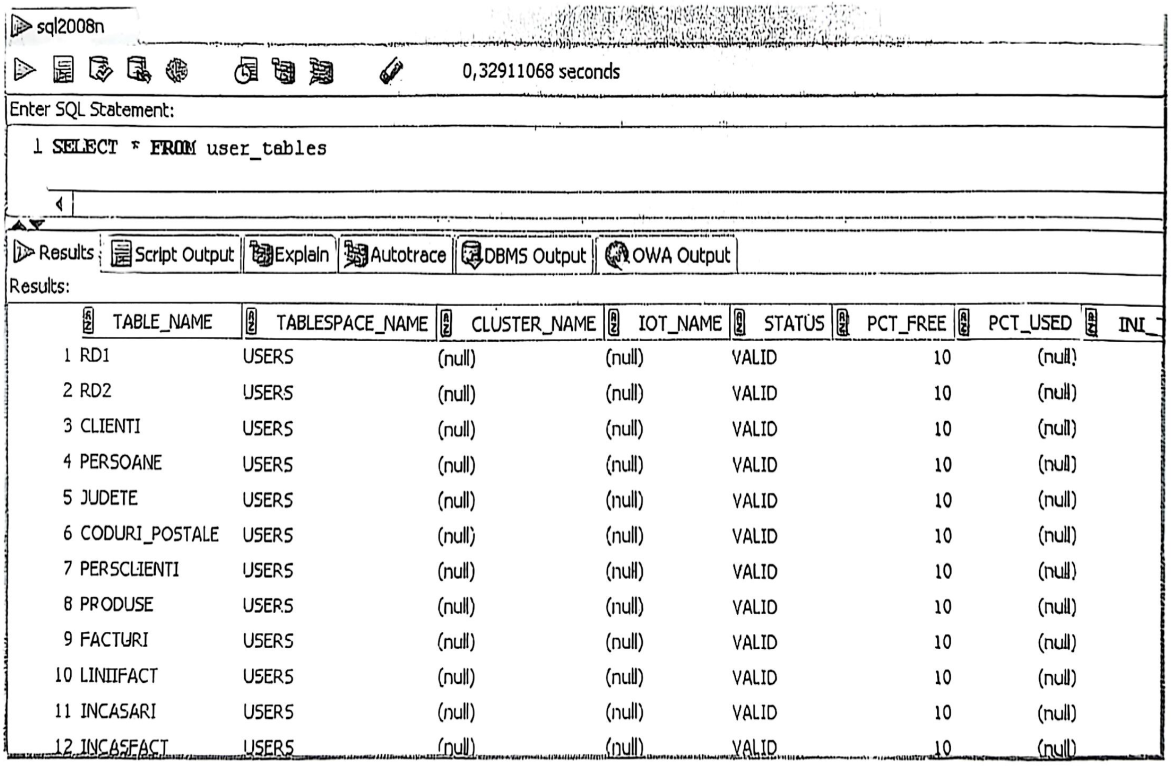
SELECT * FROM user_objects

2. Acesta este un moment publicitar.

TABLE_NAME	COMMENTS
USER_RESOURCE_LIMITS	Display resource limit of the user
USER_PASSWORD_LIMITS	Display password limits of the user
USER_CATALOG	Tables, Views, Synonvms and Sequences owned by the user
ALL_CATALOG	All tables, views, synonyms, sequences accessible to the user
USER_CLUSTERS	Descriptions of user's own clusters
ALL_CLUSTERS	Description of clusters accessible to the user
USER_CLU_COLUMNS	Mapping of table columns to cluster columns
USER_COL_COMMENTS	Comments on columns of user's tables and views
ALL_COL_COMMENTS	Comments on columns of accessible tables and views
USER_COL_PRIVS	Grants on columns for which the user is the owner, grantor or grantee
ALL_COL_PRIVS	Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
USER_COL_PRIVS_MADE	All grants on columns of objects owned by the user
ALL_COL_PRIVS_MADE	Grants on columns for which the user is owner or grantor
USER_COL_PRIVS_RECD	Grants on columns for which the user is the grantee

Figura 15.27. O porţiune din tabela virtuală „centrală" din dicţionarul bazei de date (Oracle)

Unele obiecte au informaţiile presărate în mai multe tabele virtuale. De exemplu, în USER_TABLES ne-am aştepta să găsim o sumedenie de detalii despre tabele. Aşa este (vezi figura 15.28), doar că interesaţi ar fi doar administratorii. De fapt, view-ul mai are destule atribute, dar să nu vă inchipuiţi că sunt prea multe pe care să le înţelegeţi (dacă aveţi nivelul meu de know-how (how) în Oracle).



	TABLE_NAME	TABLESPACE_NAME	CLUSTER_NAME	IOT_NAME	STATUS	PCT_FREE	PCT_USED	INI_
1	RD1	USERS	(null)	(null)	VALID	10	(null)	
2	RD2	USERS	(null)	(null)	VALID	10	(null)	
3	CLIENTI	USERS	(null)	(null)	VALID	10	(null)	
4	PERSOANE	USERS	(null)	(null)	VALID	10	(null)	
5	JUDETE	USERS	(null)	(null)	VALID	10	(null)	
6	CODURI_POSTALE	USERS	(null)	(null)	VALID	10	(null)	
7	PERSCLIENTI	USERS	(null)	(null)	VALID	10	(null)	
8	PRODUSE	USERS	(null)	(null)	VALID	10	(null)	
9	FACTURI	USERS	(null)	(null)	VALID	10	(null)	
10	LINIIFACT	USERS	(null)	(null)	VALID	10	(null)	
11	INCASARI	USERS	(null)	(null)	VALID	10	(null)	
12	INCASFACT	USERS	(null)	(null)	VALID	10	(null)	

Figura 15.28. Câteva linii şi coloane din tabela virtuală USER_TABLES

Mai mucalită este tabela virtuală USER_CATALOG, din care putem afla numai numele tuturor tabelelor, tabelelor virtuale, sinonimelor şi secvenţelor din schema utilizatorului curent:

```
SELECT * FROM user_catalog
```

Despre atributele unei tabele aflăm câte ceva din tabela USER_TAB_COLUMNS. O parte dintre coloanele acesteia (şi liniile dedicate atributelor tabelei FACTURI) este prezentată în figura 15.29.

SELECT * FROM user_tab_columns WHERE table_name = 'FACTURI'

TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_TYPE_MOD	DATA_TYPE_OWNER	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	[illegible]
FACTURI	NRFACT	NUMBER	(null)	(null)	22	8	0	N
FACTURI	DATAFACT	DATE	(null)	(null)	7	(null)	(null)	N
FACTURI	CODCL	NUMBER	(null)	(null)	22	6	0	N
FACTURI	OBS	VARCHAR2	(null)	(null)	50	(null)	(null)	Y
FACTURI	VALTOTALA	NUMBER	(null)	(null)	22	10	2	Y
FACTURI	TVA	NUMBER	(null)	(null)	22	10	2	Y
FACTURI	VALINCASATA	NUMBER	(null)	(null)	22	10	2	Y
FACTURI	REDUCERI	NUMBER	(null)	(null)	22	9	2	Y
FACTURI	PENALIZARI	NUMBER	(null)	(null)	22	9	2	Y

Figura 15.29. Câteva coloane din view-ul USER_TAB_COLUMNS dedicate atributelor tabelei FACTURI

Informaţiile sintetice despre toate restricţiile definite în tabelele aflate în schema curentă sunt disponibile în tabela virtuală *USER_CONSTRAINTS*, în care atributul *Constraint_Type* poate avea valorile P (pentru cheile primare), U (pentru cheile alternative), R (pentru restricţiile referenţiale) şi C (pentru restricţiile-utilizator la nivel de atribut şi înregistrare). Astfel, pentru a afla toate restricţiile de tip CHECK definite în tabela JUDEŢE (figura 15.30), folosim interogarea:

```
SELECT constraint_name, search_condition
FROM user_constraints
WHERE table_name = 'JUDETE' AND constraint_type = 'C'
```

CONSTRAINT_NAME	SEARCH_CONDITION
NN_JUDET	"JUDET" IS NOT NULL
NN_REGIUNE	"REGIUNE" IS NOT NULL
CK_JUD	jud=LTRIM(UPPER(jud))
CK_JUDET	judet=LTRIM(INITCAP(judet))
CK_REGIUNE	regiune IN ('Banat', 'Transilvania', 'Dobrogea', 'Oltenia', 'Muntenia', 'Moldova')

Figura 15.30. Restricţiile de tip CHECK din tabela JUDEŢE

Alcătuirea cheilor primare şi alternative este disponibilă recurgând (şi) la tabela virtuală USER_CONS_COLUMNS. Folosim o interogare similară celei din paragraful anterior (vezi figura 15.31):

```
SELECT uc.constraint_name, constraint_type, '(' || ucc1.column_name ||
        CASE WHEN ucc2.column_name IS NULL THEN '' ELSE ','||ucc2.column_name
        END ||
        CASE WHEN ucc3.column_name IS NULL THEN '' ELSE ','||ucc3.column_name
        END ||
        CASE WHEN ucc4.column_name IS NULL THEN '' ELSE ','||ucc4.column_name
        END ||')' AS expresie_restrictie
FROM user_constraints uc
```

```
        LEFT OUTER JOIN user_cons_columns ucc1
                ON uc.constraint_name = ucc1.constraint_name AND ucc1.position = 1
        LEFT OUTER JOIN user_cons_columns ucc2
                ON uc.constraint_name = ucc2.constraint_name AND ucc2.position = 2
        LEFT OUTER JOIN user_cons_columns ucc3
                ON uc.constraint_name = ucc3.constraint_name AND ucc3.position = 3
        LEFT OUTER JOIN user_cons_columns ucc4
                ON uc.constraint_name = ucc4.constraint_name AND ucc4.position = 4
WHERE uc.table_name = 'LINIIFACT' AND uc.constraint_type IN ('P', 'U')
```

CONSTRAINT_NAME	CONSTRAINT_TYPE	EXPRESIE_RESTRICTIE
PK_LINIIFACT	P	(NRFACT,LINIE)
UN_LINIIFACT	U	(NRFACT,CODPR)

Figura 15.31. Compoziţia cheilor primare şi alternative ale tabelei LINIIFACT

Încheiem paragraful cu acelaşi exemplu din PostgreSQL, dedicat recompunerii restricţiilor referenţiale. Oracle nu permite clauza ON UPDATE CASCADE, iar ON DELETE nu ne interesează, aşa că efortul redactării comenzii SELECT care să afişeze informaţii despre cheile străine declarate în tabela LINIIFACT (figura 15.32) este ceva mai redus:

```
SELECT uc.constraint_name AS restrictie_copil, ucc1.table_name AS tabela_copil,
        ucc1.column_name AS atribut_copil,
        r_constraint_name AS restrictie_parinte, ucc2.table_name AS tabela_parinte,
        ucc2.column_name AS atribut_parinte
FROM user_constraints uc
        INNER JOIN user_cons_columns ucc1
                ON uc.constraint_name = ucc1.constraint_name
        INNER JOIN user_cons_columns ucc2
                ON uc.r_constraint_name = ucc2.constraint_name
                        AND ucc1.position = ucc2.position
WHERE uc.table_name = 'LINIIFACT' AND uc.constraint_type = 'R'
ORDER BY uc.constraint_name, ucc1.position
```

RESTRICTIE_COPIL	TABELA_COPIL	ATRIBUT_COPIL	RESTRICTIE_PARINTE	TABELA_PARINTE	ATRIBUT_PARINTE
FK_LINIIFACT_FACTURI	LINIIFACT	NRFACT	PK_FACTURI	FACTURI	NRFACT
FK_LINIIFACT_PRODUSE	LINIIFACT	CODPR	PK_PRODUSE	PRODUSE	CODPR

Figura 15.32. Afişarea datelor despre cheile străine ale tabelei LINIIFACT

15.2.3. Obiecte din schema unei baze de date în DB2

DB2 administrează două seturi de tabele virtuale în cadrul dicţionarului de date, în schemele SYSCAT şi SYSSTAT. Ambele seturi sunt accesibile utilizatorilor.

Tabelele virtuale din SYSSTAT reprezintă un subset al celor din SYSCAT, singurul avantaj fiind că o parte din atribute sunt actualizabile, deşi acest lucru este foarte riscant. Cei de la IBM au mai introdus câteva view-uri în schema SYSIBM şi pentru a se apropia de prevederile standardului. Date despre tabelele virtuale (figura 15.33) dicţionar pot fi obţinute simplu:

SELECT * FROM syscat.views

VIEWSCHEMA	VIEWNAME	OWNER	OWNERTYPE	SEQ...	VIEWCHECK	READONLY	VALID	QUA
SYSIBM	CHECK_CONSTRAINTS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	COLUMNS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	COLUMNS_S	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	DUAL	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	REFERENTIAL_CONSTRAINTS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	REF_CONSTRAINTS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	TABLE_CONSTRAINTS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	TABLES	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	TABLES_S	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	USER_DEFINED_TYPES	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	UDT_S	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	VIEWS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	PARAMETERS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	PARAMETERS_S	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	ROUTINES	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	ROUTINES_S	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	SYSFUNCTIONS	SYSIBM	S	1	N	N	Y	SYSI
SYSIBM	SYSPROCEDURES	SYSIBM	S	1	N	N	Y	SYSI
SYSIBM	SYSFUNCPARMS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	SYSPROCPARMS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	SYSREVTYPEMAPPINGS	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	SYSDUMMY1	SYSIBM	S	1	N	Y	Y	SYSI
SYSIBM	SYSROUTINEPROPERTIESJAVA	SYSIBM	S	1	N	Y	Y	SYSI
SYSCAT	ATTRIBUTES	SYSIBM	S	1	N	Y	Y	SYSI
SYSCAT	AUDITPOLICIES	SYSIBM	S	1	N	Y	Y	SYSI
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figura 15.33. Câteva informaţii despre (câteva) view-uri dicţionar

Cele două scheme din care se afişeaza view-urile sunt SYSIBM şi SYSCAT (figura ilustrează doar o parte din rezultat). Numele tabelelor sunt destul de sugestive: TABLES, COLUMNS, CHECK_CONSTRAINTS etc. Spre exemplu, ne interesează să aflăm numele tabelelor din schema curentă. Folosind view-ul TABLES din schema SYSCATE, interogarea este:

SELECT tabname FROM SYSCAT.tables

WHERE tabschema = CURRENT_USER AND type = 'T'

ORDER BY 1

iar folosind view-ul cu acelaşi nume, dar din (sub)schema SYSIBM, sintaxa este:

SELECT table_name FROM SYSIBM.tables

WHERE table_schema = CURRENT_USER AND table_type = 'BASE TABLE'

ORDER BY 1

Câteva informaţii despre atributele tabelei FACTURI pot fi aflate prin interogarea ce foloseşte *syscat.columns* (vezi figura 15.34):

SELECT colname, colno, typename, length, scale, default, nulls

FROM SYSCAT.columns

WHERE tabschema = CURRENT_USER AND tabname = 'FACTURI'

ORDER BY colno

COLNAME	COLNO	TYPENAME	LENGTH	SCALE	DEFAULT	NULLS
NRFACT	0	INTEGER	4	0		N
DATAFACT	1	DATE	4	0	CURRENT DATE	N
CODCL	2	SMALLINT	2	0		N
OBS	3	VARCHAR	50	0		Y
VALTOTALA	4	DECIMAL	10	2	0	Y
TVA	5	DECIMAL	10	2	0	Y
VALINCASATA	6	DECIMAL	10	2	0	Y
REDUCERI	7	DECIMAL	9	2	0	Y
PENALIZARI	8	DECIMAL	9	2	0	Y

Figura 15.34. Informații despre atributele tabelei FACTURI furnizate de *syscat.columns*

Alte detalii, dintre care doar o parte sunt cuprinse în figura 15.35, sunt obținute prin consultarea *sysibm.columns*:

TABL...	TA...	TABL...	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NUL...	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH	CHAR...
SQL2008	MARIN	FACTURI	CODCL	3		NO	SMALLINT		
SQL2008	MARIN	FACTURI	DATAFACT	2	CURRENT DATE	NO	DATE		
SQL2008	MARIN	FACTURI	NRFACT	1		NO	INTEGER		
SQL2008	MARIN	FACTURI	OBS	4		YES	CHARACTER ...	50	
SQL2008	MARIN	FACTURI	PENALIZARI	9	0	YES	DECIMAL		
SQL2008	MARIN	FACTURI	REDUCERI	8	0	YES	DECIMAL		
SQL2008	MARIN	FACTURI	TVA	6	0	YES	DECIMAL		
SQL2008	MARIN	FACTURI	VALINCASATA	7	0	YES	DECIMAL		
SQL2008	MARIN	FACTURI	VALTOTALA	5	0	YES	DECIMAL		

Figura 15.35. Informații despre atributele tabelei FACTURI furnizate de *sysibm.columns*

Restricțiile definite în tabelele unei scheme constituie obiectul tabelei virtuale TABCONST din schema *syscat*. Iată cum le obținem pe cele declarate în tabele JUDEȚE, FACTURI, LINIIFACT, DOCTORI şi GĂRZI (vezi figura 15.36):

CONSTNAME	TABNAME	TYPE
PK_DOCTORI	DOCTORI	P
FK_FACTURI_CLIENTI	FACTURI	F
CK_DATAFACT	FACTURI	K
PK_FACTURI	FACTURI	P
SQL081009123234650	GARZI	F
PK_GARZI	GARZI	P
CK_JUD	JUDETE	K
CK_REGIUNE	JUDETE	K
PK_JUDETE	JUDETE	P
NN_REGIUNE	JUDETE	U
FK_LINIIFACT_FACTURI	LINIIFACT	F
FK_LINIIFACT_PRODUSE	LINIIFACT	F
CK_LINIE	LINIIFACT	K
PK_LINIIFACT	LINIIFACT	P
UN_LINIIFACT	LINIIFACT	U

Figura 15.36. Restricții în câteva tabele

```
SELECT constname, tabname, type FROM SYSCAT.tabconst
WHERE tabschema = CURRENT_USER AND
        tabname IN ('JUDETE', 'FACTURI', 'LINIIFACT', 'DOCTORI', 'GARZI')
ORDER BY tabname, type, constname
```

Pentru că nu am folosit clauza CONSTRAINT la declararea restricţiei referenţiale între tabela GĂRZI (copil) şi DOCTORI (părinte), numele atribuit „din burtă" acestei restricţii este destul de lung - SQL08010091...

Am crede că (aproximativ) acelaşi rezultat l-am obţine dacă am recurge şi la view-ul TABLE_CONSTRAINTS din schema SYSIBM, ştiind că numele atributelor sunt un pic schimbate:

```
SELECT constraint_name, table_name, constraint_type
FROM sysibm.table_constraints
WHERE table_schema = CURRENT_USER AND
        table_name IN ('JUDETE', 'FACTURI', 'LINIIFACT',
                'DOCTORI', 'GARZI')
ORDER BY table_name, constraint_type, constraint_name
```

Figura 15.37 ne înşală aşteptările, numărul restricţiilor furnizate fiind sensibil mai mare. Apar în plus o serie de CHECK-uri pe care nu ne amintim să le fi declarat în scripturile de creare sau ulterior. Fiecare dintre cele patru tabele este „împroprietărită" cu câteva CHECK-uri, deci sistemul pare destul de democratic. Vom vedea îndată explicaţia.

CONSTRAINT_NAME	TABLE_NAME	CONSTRAINT_TYPE
PK_DOCTORI	DOCTORI	PRIMARY KEY
1009123234465001000005000200	DOCTORI	CHECK
1009123234465001010005000200	DOCTORI	CHECK
FK_FACTURI_CLIENTI	FACTURI	FOREIGN KEY
PK_FACTURI	FACTURI	PRIMARY KEY
110308195870300500000F000200	FACTURI	CHECK
110308195870300501000F000200	FACTURI	CHECK
110308195870300502000F000200	FACTURI	CHECK
CK_DATAFACT	FACTURI	CHECK
SQL081009123234650	GARZI	FOREIGN KEY
PK_GARZI	GARZI	PRIMARY KEY
1009123234637001000006000200	GARZI	CHECK
1009123234637001010006000200	GARZI	CHECK
PK_JUDETE	JUDETE	PRIMARY KEY
NN_REGIUNE	JUDETE	UNIQUE
1103081957500001000009000200	JUDETE	CHECK
1103081957500001010009000200	JUDETE	CHECK
1103081957500001020009000200	JUDETE	CHECK
CK_JUD	JUDETE	CHECK
CK_REGIUNE	JUDETE	CHECK
FK_LINIIFACT_FACTURI	LINIIFACT	FOREIGN KEY
FK_LINIIFACT_PRODUSE	LINIIFACT	FOREIGN KEY
PK_LINIIFACT	LINIIFACT	PRIMARY KEY
UN_LINIIFACT	LINIIFACT	UNIQUE
1103081958812001000010000200	LINIIFACT	CHECK
1103081958812001010010000200	LINIIFACT	CHECK
1103081958812001020010000200	LINIIFACT	CHECK
1103081958812001030010000200	LINIIFACT	CHECK
1103081958812001040010000200	LINIIFACT	CHECK
CK_LINIE	LINIIFACT	CHECK

Figura 15.37. *Sysibm.table_constraints* furnizează mai multe restricţii decât *syscat.tabconst*

Restricţiile-utilizator la nivel de atribut şi înregistrare pot fi aflate din tabela virtuală CHECKS, din schema *syscat* (vezi figura 15.38). Au fost extrase numele restricţiei, numele tabelei şi clauza CHECK efectivă:

```
SELECT constname, tabname, text
FROM SYSCAT.checks
WHERE tabschema = CURRENT_USER
```

CONSTNAME	TABNAME	TEXT
CK_ADRESA_C...	CLIENTI	SUBSTR(ADRESA,1,1) = UPPER(SUBSTR(ADRESA,1,1))
CK_CODCL	CLIENTI	CODCL > 1000
CK_CODFISCAL	CLIENTI	SUBSTR(CODFISCAL,1,1) = UPPER(SUBSTR(CODFISCAL,1,1))
CK_DENCL	CLIENTI	SUBSTR(DENCL,1,1) = UPPER(SUBSTR(DENCL,1,1))
NN_LOC	CODURI_POS...	CODPOST=LTRIM(CODPOST)
CK_DATAFACT	FACTURI	DATAFACT >= '2007-01-01' AND DATAFACT <= '2015-12-31'
CK_CODDOC	INCASARI	CODDOC=UPPER(LTRIM(CODDOC))
CK_DATADOC	INCASARI	DATADOC >= '2007-01-01' AND DATADOC <= '2010-12-31'
CK_DATAINC	INCASARI	DATAINC >= '2007-01-01' AND DATAINC <= '2015-12-31'
CK_DATE_INC...	INCASARI	DATAINC >= DATADOC
CK_JUD	JUDETE	JUD=LTRIM(UPPER(JUD))
CK_REGIUNE	JUDETE	REGIUNE IN ('Banat', 'Transilvania', 'Dobrogea', 'Oltenia', 'Muntenia', 'Moldova')
CK_LINIE	LINIIFACT	LINIE > 0
CK_FUNCTIE	PERSCLIENTI	SUBSTR(FUNCTIE,1,1) = UPPER(SUBSTR(FUNCTIE,1,1))
CK_ADRESA_P...	PERSOANE	SUBSTR(ADRESA,1,1) = UPPER(SUBSTR(ADRESA,1,1))
CK_CNP	PERSOANE	CNP=LTRIM(UPPER(CNP))
CK_SEX	PERSOANE	SEX IN ('F','B')
CK_CODPR	PRODUSE	CODPR > 0
CK_DENPR	PRODUSE	SUBSTR(DENPR,1,1) = UPPER(SUBSTR(DENPR,1,1))
CK_PRODUSE...	PRODUSE	SUBSTR(GRUPA,1,1) = UPPER(SUBSTR(GRUPA,1,1))

Figura 15.38. Restricţii de tip CHECK furnizate de *syscat.checks*

Numele fiecărui atribut implicat într-un CHECK este disponibil dintr-o altă tabelă virtuală a dicţionarului - *syscat.COLCHECKS*.

Dacă dorim să comparăm rezultatul cu cel furnizat de view-ul SYSIBM. CHECK_CONSTRAINTS, ne aşteptăm ca, după exemplul precedent, să apară câteva linii în plus. Surpriză: este identic!

```
SELECT cc.constraint_name, table_name, check_clause
FROM sysibm.check_constraints cc INNER JOIN sysibm.table_constraints tc
        ON cc.constraint_name = tc.constraint_name
WHERE table_schema = CURRENT_USER
ORDER BY table_name, constraint_name
```

Să recapitulăm ceea ce am observat în ultimele două exemple folosind view-uri din schema SYSIBM. TABLE_CONSTRAINTS furnizează nişte restricţii (cu nume dubioase) de tip CHECK care însă nu se găsesc în CHECK_CONSTRAINTS. Să le extragem pe cele din tabelele JUDEŢE, LINIIFACT, DOCTORI şi GĂRZI (figura 15.39):

```
SELECT constraint_name, table_name, constraint_type
FROM sysibm.table_constraints
WHERE table_schema = CURRENT_USER AND table_name IN
```

```
('JUDETE', 'FACTURI', 'LINIIFACT', 'DOCTORI', 'GARZI')
    AND constraint_type = 'CHECK' AND constraint_name NOT IN
        (SELECT constraint_name
         FROM sysibm.check_constraints )
ORDER BY table_name, constraint_type, constraint_name
```

În figură, pentru tabela JUDEŢE sunt trei linii. Folosindu-ne de intuiţia feminină, putem concluziona că aceste restricţii sunt, de fapt, NOT NULL-uri pe care DB2 le consideră de tip CHECK, de la sine putere.

CONSTRAINT_NAME	TABLE_NAME	CONSTRAINT_TYPE
10091232344650010000005000200	DOCTORI	CHECK
10091232344650010100005000200	DOCTORI	CHECK
110308195870300500000F000200	FACTURI	CHECK
110308195870300501000F000200	FACTURI	CHECK
110308195870300502000F000200	FACTURI	CHECK
10091232346370010000006000200	GARZI	CHECK
10091232346370010100006000200	GARZI	CHECK
11030819575000010000009000200	JUDETE	CHECK
11030819575000010100009000200	JUDETE	CHECK
11030819575000010200009000200	JUDETE	CHECK
11030819588120010000010000200	LINIIFACT	CHECK
11030819588120010100010000200	LINIIFACT	CHECK
11030819588120010200010000200	LINIIFACT	CHECK
11030819588120010300010000200	LINIIFACT	CHECK
11030819588120010400010000200	LINIIFACT	CHECK

Figura 15.39. Restricţii de tip CHECK „mincinoase”

Analog paragrafelor anterioare, ne propunem să aflăm compoziţia cheilor primare, alternative şi străine din tabela LINIIFACT. Iată una dintre soluţiile DB2:

```
SELECT tc.constname, '(' || kcu1.colname ||
        CASE WHEN kcu2.colname IS NULL THEN '' ELSE ','||kcu2.colname END
        || CASE WHEN kcu3.colname IS NULL THEN '' ELSE ','||kcu3.colname END
        || CASE WHEN kcu4.colname IS NULL THEN '' ELSE ','||kcu4.colname END
            ||')' AS expresie_restrictie
FROM SYSCAT.tabconst tc
        INNER JOIN SYSCAT.keycoluse kcu1 ON tc.constname=kcu1.constname
            AND kcu1.colseq = 1
        LEFT OUTER JOIN SYSCAT.keycoluse kcu2 ON tc.constname=kcu2.constname
            AND kcu2.colseq = 2
        LEFT OUTER JOIN SYSCAT.keycoluse kcu3 ON tc.constname=kcu3.constname
            AND kcu3.colseq = 3
        LEFT OUTER JOIN SYSCAT.keycoluse kcu4 ON tc.constname=kcu4.constname
            AND kcu4.colseq = 4
WHERE tc.tabschema = CURRENT_USER AND tc.tabname = 'LINIIFACT'
```

Cât despre restricţiile referenţiale, varianta pe care o prezentăm pentru afişarea clauzelor FOREIGN KEY din tabela LINIIFACT (figura 15.40) este neverosimil de

simplă, bazându-se pe o singură tabelă virtuală - SYSCAT.REFERENCES (şi pe faptul că o cheie străină este simplă):

```
SELECT ' CONSTRAINT ' || constname || ' FOREIGN KEY (' || RTRIM(fk_colnames)
        || ') REFERRENCES ' || reftabname || ' (' || RTRIM(fk_colnames) || ')'
FROM SYSCAT.references
WHERE tabschema = CURRENT_USER AND tabname = 'LINIIFACT'
```

EXPRESIE_FOREIGN_KEY
CONSTRAINT FK_LINIIFACT_FACTURI FOREIGN KEY (NRFACT) REFERRENCES FACTURI (NRFACT)
CONSTRAINT FK_LINIIFACT_PRODUSE FOREIGN KEY (CODPR) REFERRENCES PRODUSE (CODPR)

Figura 15.40. Clauzele FOREIGN KEY pentru tabela LINIIFACT

15.2.4. Obiecte din schema unei baze de date în MS SQL Server

SQL Server are un triplu mecanism de furnizare a informaţiilor despre schema BD. Primul, moştenit de pe „vremea Sybase" ([Kriegel & Trukhnov 2008], p. 511), este bazat pe proceduri stocate speciale şi nu ne interesează deloc (cel puţin pe mine). Al doilea este un mecanism propriu de tabele şi tabele virtuale creat în schema SYS, iar al treilea se doreşte o aliniere la prevederile standardului, fiind implementat prin tabele virtuale plasate în schema INFORMATION_SCHEMA.

LAPTOPD.SQL20...QLQuery1.sql* | Summary

SELECT * FROM INFORMATION_SCHEMA.SCHEMATA

Results | Messages

	CATALOG_NAME	SCHEMA_NAME	SCHEMA_OWNER	DEFAULT_CH
1	SQL2008	dbo	dbo	NULL
2	SQL2008	guest	guest	NULL
3	SQL2008	INFORMATION_SCHEMA	INFORMATION_SCHEMA	NULL
4	SQL2008	sys	sys	NULL
5	SQL2008	salarizare	salarizare	NULL
6	SQL2008	db_owner	db_owner	NULL
7	SQL2008	db_accessadmin	db_accessadmin	NULL
8	SQL2008	db_securityadmin	db_securityadmin	NULL
9	SQL2008	db_ddladmin	db_ddladmin	NULL
10	SQL2008	db_backupoperator	db_backupoperator	NULL
11	SQL2008	db_datareader	db_datareader	NULL
12	SQL2008	db_datawriter	db_datawriter	NULL
13	SQL2008	db_denydatareader	db_denydatareader	NULL
14	SQL2008	db_denydatawriter	db_denydatawriter	NULL

Figura 15.41. Subscheme din BD curentă (în SQL Server)

Astfel, pentru a vizualiza toate (sub)schemele existente în baza de date, interogarea se scrie astfel:

```
SELECT * FROM INFORMATION_SCHEMA.SCHEMATA
```

Din figura 15.41, recunoaştem doar trei scheme, *dbo* - cea în care am creat tabelele de până acum, *salarizare* - cea creată în acest capitol şi nou-discutata *INFORMATION_SCHEMA*.

Un exemplu de view-dicţionar non-standard este *objects* care conţine informaţii sintetice despre toate obiectele din subschemele la care utilizatorul curent are drepturi de acces (figura 15.42):

SELECT * FROM sys.objects

LAPTOPD.SQL20...QLQuery1.sql* | Summary

```
SELECT * FROM sys.objects
```

Results | Messages

	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
41	sysbinsubobjs	97	NULL	4	0	S	SYSTEM_TABLE	2005-10-14 01:36:22.110	2005-10-14
42	personal2	289436105	NULL	1	0	U	USER_TABLE	2008-10-20 11:59:30.153	2008-10-20
43	pk_personal2	305436162	NULL	1	289436105	PK	PRIMARY_KEY_CONSTRAINT	2008-10-20 11:59:30.153	2008-10-20
44	fk_personal2	321436219	NULL	1	289436105	F	FOREIGN_KEY_CONSTRAINT	2008-10-20 11:59:30.153	2008-10-20
45	sporuri	337436276	NULL	1	0	U	USER_TABLE	2008-10-20 11:59:30.153	2008-10-20
46	pk_sporuri	353436333	NULL	1	337436276	PK	PRIMARY_KEY_CONSTRAINT	2008-10-20 11:59:30.153	2008-10-20
47	fk_sporuri_pers2	369436390	NULL	1	337436276	F	FOREIGN_KEY_CONSTRAINT	2008-10-20 11:59:30.153	2008-10-20
48	rd1	417436561	NULL	1	0	U	USER_TABLE	2008-10-22 08:08:49.293	2008-10-22
49	rd2	433436618	NULL	1	0	U	USER_TABLE	2008-10-22 08:08:49.310	2008-10-22
50	judete	449436675	NULL	1	0	U	USER_TABLE	2008-10-23 15:22:36.013	2008-10-23
51	pk_judete	465436732	NULL	1	449436675	PK	PRIMARY_KEY_CONSTRAINT	2008-10-23 15:22:36.013	2008-10-23
52	un_judet	481436789	NULL	1	449436675	UQ	UNIQUE_CONSTRAINT	2008-10-23 15:22:36.013	2008-10-23
53	ck_jud	497436846	NULL	1	449436675	C	CHECK_CONSTRAINT	2008-10-23 15:22:36.013	2008-10-23
54	DF__judete__R...	513436903	NULL	1	449436675	D	DEFAULT_CONSTRAINT	2008-10-29 15:22:36.013	2008-10-23
55	ck_regiune	529436960	NULL	1	449436675	C	CHECK_CONSTRAINT	2008-10-23 15:22:36.013	2008-10-23
56	coduri_postale	545437017	NULL	1	0	U	USER_TABLE	2008-10-23 15:22:36.013	2008-10-23

Figura 15.42. O mică parte din conţinutul tabelei virtuale SYS.OBJECTS

Ca şi în DB2, putem formula interogări atât asupra view-urilor din schema SYS, cât şi a celor din schema INFORMATION_SCHEMA. De exemplu, putem afla numele tabelelor create de noi în schema *dbo* fie cu interogarea:

```
SELECT name
FROM sys.tables
WHERE type_desc = 'USER_TABLE' AND schema_id IN
        (SELECT schema_id FROM sys.schemas WHERE name = 'dbo')
ORDER BY 1
```

fie prin:

```
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'dbo' AND TABLE_TYPE = 'BASE TABLE'
ORDER BY 1
```

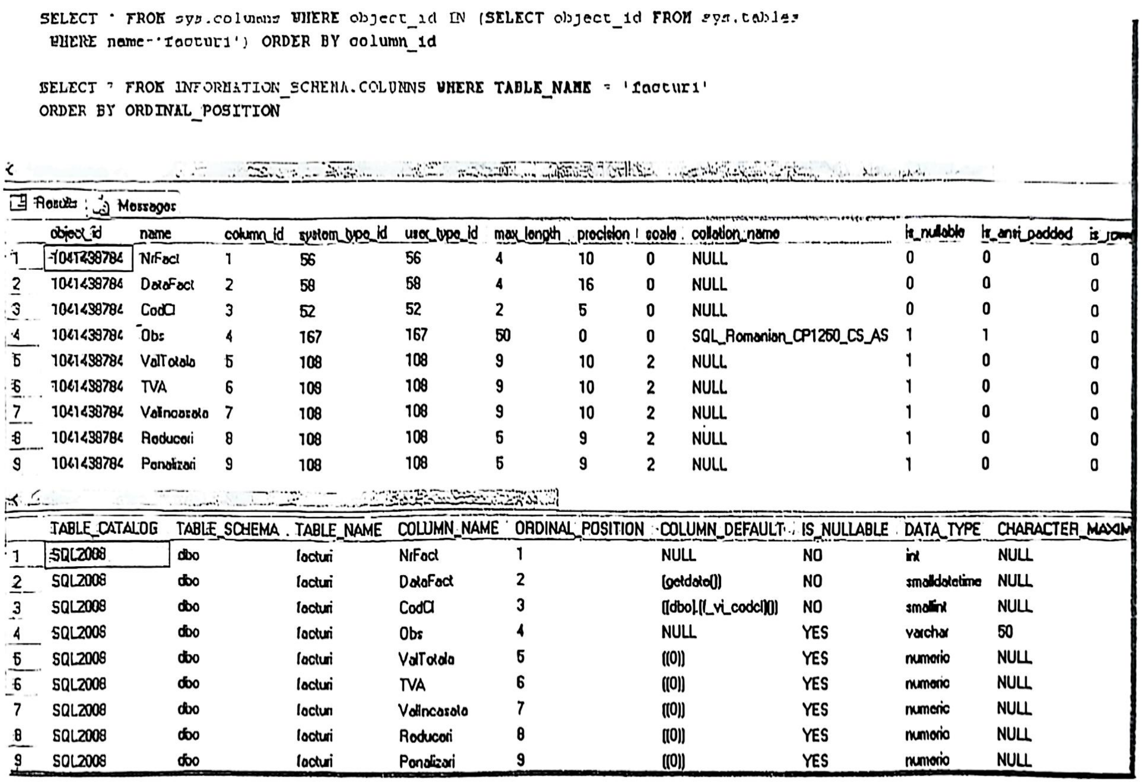
Informaţiile despre atribute sunt plasate în tabelele virtuale *sys.columns* şi *INFORMATION_SCHEMA.COLUMNS*. Astfel, aflăm date despre atributele tabelei FACTURI (vezi figura 15.43) fie prin varianta:

```
SELECT *
FROM sys.columns
WHERE object_id IN (SELECT object_id FROM sys.tables
```

```
                WHERE name = 'facturi')
ORDER BY column_id
```

fie prin:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'facturi'
ORDER BY ORDINAL_POSITION
```



```
SELECT * FROM sys.columns WHERE object_id IN (SELECT object_id FROM sys.tables
 WHERE name='facturi') ORDER BY column_id

SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'facturi'
ORDER BY ORDINAL_POSITION
```

Results | Messages

	object_id	name	column_id	system_type_id	user_type_id	max_length	precision	scale	collation_name	is_nullable	is_ansi_padded	is_...
1	1041438784	NrFact	1	56	56	4	10	0	NULL	0	0	0
2	1041438784	DataFact	2	58	58	4	16	0	NULL	0	0	0
3	1041438784	CodCl	3	52	52	2	5	0	NULL	0	0	0
4	1041438784	Obs	4	167	167	50	0	0	SQL_Romanian_CP1250_CS_AS	1	1	0
5	1041438784	ValTotala	5	108	108	9	10	2	NULL	1	0	0
6	1041438784	TVA	6	108	108	9	10	2	NULL	1	0	0
7	1041438784	ValIncasata	7	108	108	9	10	2	NULL	1	0	0
8	1041438784	Reduceri	8	108	108	5	9	2	NULL	1	0	0
9	1041438784	Penalizari	9	108	108	5	9	2	NULL	1	0	0

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIM...
1	SQL2008	dbo	facturi	NrFact	1	NULL	NO	int	NULL
2	SQL2008	dbo	facturi	DataFact	2	(getdate())	NO	smalldatetime	NULL
3	SQL2008	dbo	facturi	CodCl	3	([dbo].[f_vi_codcl]())	NO	smallint	NULL
4	SQL2008	dbo	facturi	Obs	4	NULL	YES	varchar	50
5	SQL2008	dbo	facturi	ValTotala	5	((0))	YES	numeric	NULL
6	SQL2008	dbo	facturi	TVA	6	((0))	YES	numeric	NULL
7	SQL2008	dbo	facturi	ValIncasata	7	((0))	YES	numeric	NULL
8	SQL2008	dbo	facturi	Reduceri	8	((0))	YES	numeric	NULL
9	SQL2008	dbo	facturi	Penalizari	9	((0))	YES	numeric	NULL

Figura 15.43. Două variante de vizualizare a informaţiilor despre atributele tabelei FACTURI

Valorile implicite ale atributelor pot fi vizualizate dintr-un view special - *sys.default_constraints* - sau tot din *INFORMATION_SCHEMA.COLUMNS*.

Aflarea restricţiilor definite în tabelele JUDEŢE, FACTURI, LINIIFACT şi GĂRZI (vezi figura 15.44) presupun folosirea unei interogări aproape identice celei din DB2[3]:

```
SELECT constraint_name, table_name, constraint_type
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
WHERE table_schema = CURRENT_USER AND
        table_name IN ('judete', 'facturi', 'liniifact', 'doctori', 'garzi')
ORDER BY table_name, constraint_type, constraint_name
```

3. Numele celor cinci tabele se scriu cu minuscule (aşa le-am creat în capitolul 3).

constraint_name	table_name	constraint_type
pk_doctori	doctori	PRIMARY KEY
ck_datafact	facturi	CHECK
ck_facturi_codcl	facturi	CHECK
fk_facturi_clienti	facturi	FOREIGN KEY
pk_facturi	facturi	PRIMARY KEY
FK__garzi__iddoctor__581D0306	garzi	FOREIGN KEY
pk_garzi	garzi	PRIMARY KEY
ck_jud	judete	CHECK
ck_regiune	judete	CHECK
pk_judete	judete	PRIMARY KEY
un_judet	judete	UNIQUE
ck_linie	liniifact	CHECK
fk_liniifact_facturi	liniifact	FOREIGN KEY
fk_liniifact_produse	liniifact	FOREIGN KEY
pk_liniifact	liniifact	PRIMARY KEY

Figura 15.44. Restricţii definite în cinci tabele

Expresiile clauzelor CHECK ale celor patru tabele se obţin prin varianta:

```
SELECT table_name, tc.constraint_name, check_clause
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
        INNER JOIN INFORMATION_SCHEMA.CHECK_CONSTRAINTS cc
                ON tc.constraint_name = cc.constraint_name
WHERE table_schema = CURRENT_USER AND
        table_name IN ('judete', 'facturi', 'liniifact',
                'doctori', 'garzi') AND constraint_type = 'CHECK'
        ORDER BY table_name, tc.constraint_name
```

sau:

```
SELECT o2.name AS table_name, cc.name AS constraint_name,
        cc.definition AS check_clause
FROM sys.check_constraints cc
        INNER JOIN sys.objects o1 ON cc.name = o1.name
        INNER JOIN sys.objects o2 ON cc.parent_object_id = o2.object_id
                AND o2.name IN ('judete', 'facturi', 'liniifact', 'doctori', 'garzi')
```

Pentru un pic de variaţiune, vom extrage expresiile cheilor primare şi alternative ale tuturor tabelelor din schema curentă. Varianta de mai jos, al cărei rezultat este cuprins în figura 15.45, foloseşte tabelele virtuale-dicţionar: SYS_TABLES, SYS.KEY_CONSTRAINTS, SYS.INDEX_COLUMNS şi SYS.COLUMNS:

```
SELECT t.name AS table_name, k.name AS constraint_name,
        k.type_desc AS constraint_type,
        CASE WHEN x1.name IS NULL THEN '' ELSE '(' + x1.name END +
        CASE WHEN x2.name IS NULL THEN '' ELSE ',' + x2.name END +
```

```
        CASE WHEN x3.name IS NULL THEN '' ELSE ',' + x3.name END +
        CASE WHEN x4.name IS NULL THEN '' ELSE ',' + x4.name END +
        CASE WHEN x1.name IS NOT NULL OR x2.name IS NOT NULL OR
                x3.name IS NOT NULL OR x4.name IS NOT NULL
                        THEN ')' ELSE ',' END AS expresie_restrictie
FROM sys.tables t
        INNER JOIN sys.key_constraints k ON t.object_id = k.parent_object_id
        LEFT OUTER JOIN
                (SELECT i.object_id, index_id, c.name, key_ordinal
                FROM sys.index_columns i
                        INNER JOIN sys.columns c ON i.object_id = c.object_id
                                AND i.column_id = c.column_id
                ) x1 ON x1.object_id = t.object_id AND x1.index_id = k.unique_index_id
                                AND x1.key_ordinal = 1
        LEFT OUTER JOIN
                (SELECT i.object_id, index_id, c.name, key_ordinal
                FROM sys.index_columns i
                        INNER JOIN sys.columns c ON i.object_id = c.object_id AND
                                i.column_id = c.column_id
                ) x2 ON x2.object_id = t.object_id AND x2.index_id = k.unique_index_id
                                AND x2.key_ordinal = 2
        LEFT OUTER JOIN
                (SELECT i.object_id, index_id, c.name, key_ordinal
                FROM sys.index_columns i
                        INNER JOIN sys.columns c ON i.object_id = c.object_id AND
                                i.column_id = c.column_id
                ) x3 ON x3.object_id = t.object_id AND x3.index_id = k.unique_index_id
                        AND x3.key_ordinal = 3
        LEFT OUTER JOIN
                (SELECT i.object_id, index_id, c.name, key_ordinal
                FROM sys.index_columns i
                        INNER JOIN sys.columns c ON i.object_id = c.object_id AND
                                i.column_id = c.column_id
                ) x4 ON x4.object_id = t.object_id AND x4.index_id = k.unique_index_id
                        AND x4.key_ordinal = 4
ORDER BY 1
```

table_name	constraint_name	constraint_type	expresie_restrictie
candidati	PK__candidati__7251D655	PRIMARY_KEY_CONSTRAINT	(IdCandidat)
clienti	pk_clienti	PRIMARY_KEY_CONSTRAINT	(CodCl)
clienti	un_codfiscal	UNIQUE_CONSTRAINT	(CodFiscal)
coduri_postale	pk_coduri_post	PRIMARY_KEY_CONSTRAINT	(CodPost)
distante	PK__distante__55EAA1D1	PRIMARY_KEY_CONSTRAINT	(Loc1,Loc2)
doctori	pk_doctori	PRIMARY_KEY_CONSTRAINT	(iddoctor)
facturi	pk_facturi	PRIMARY_KEY_CONSTRAINT	(NrFact)
garzi	pk_garzi	PRIMARY_KEY_CONSTRAINT	(iddoctor,inceput_garda)
incasari	pk_incasari	PRIMARY_KEY_CONSTRAINT	(CodInc)
incasfact	pk_incasfact	PRIMARY_KEY_CONSTRAINT	(CodInc,NrFact)
judete	pk_judete	PRIMARY_KEY_CONSTRAINT	(Jud)
judete	un_judet	UNIQUE_CONSTRAINT	(Judet)
liniifact	pk_liniifact	PRIMARY_KEY_CONSTRAINT	(NrFact,Linie)
mastere	PK__mastere__70698DE3	PRIMARY_KEY_CONSTRAINT	(Spec)
pacienti	pk_pacienti	PRIMARY_KEY_CONSTRAINT	(idpacient)
persclienti	pk_persclienti	PRIMARY_KEY_CONSTRAINT	(CNP,CodCl,Functie)
persoane	pk_persoane	PRIMARY_KEY_CONSTRAINT	(CNP)
personal2	pk_personal2	PRIMARY_KEY_CONSTRAINT	(Marca)
produse	pk_produse	PRIMARY_KEY_CONSTRAINT	(CodPr)
sporuri	pk_sporuri	PRIMARY_KEY_CONSTRAINT	(An,Luna,Marca)
triaj	pk_internari	PRIMARY_KEY_CONSTRAINT	(idexaminare)

Figura 15.45. Chei primare şi alternative

Acelaşi raport se obţine mai simplu prin utilizarea doar a tabelelor virtuale din INFORMATION_SCHEMA, TABLE_CONSTRAINTS şi KEY_COLUMN_USAGE:

```
SELECT t.table_name, t.constraint_name, t.constraint_type,
  CASE WHEN k1.column_name IS NULL THEN '' ELSE '(' + k1.column_name END +
  CASE WHEN k2.column_name IS NULL THEN '' ELSE ',' + k2.column_name END +
  CASE WHEN k3.column_name IS NULL THEN '' ELSE ',' + k3.column_name END +
  CASE WHEN k4.column_name IS NULL THEN '' ELSE ',' + k4.column_name END +
  CASE WHEN k1.column_name IS NOT NULL OR k2.column_name IS NOT NULL OR
                k3.column_name IS NOT NULL OR k4.column_name IS NOT NULL
        THEN ')' ELSE ',' END AS expresie_restrictie
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS t
        LEFT OUTER JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE k1
                ON t.table_name = k1.table_name AND
                        t.constraint_name = k1.constraint_name
                        AND k1.ordinal_position = 1
        LEFT OUTER JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE k2
                ON t.table_name = k2.table_name AND
                        t.constraint_name = k2.constraint_name
                        AND k2.ordinal_position = 2
```

```
        LEFT OUTER JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE k3
                ON t.table_name = k3.table_name AND
                        t.constraint_name = k3.constraint_name
                        AND k3.ordinal_position = 3
        LEFT OUTER JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE k4
                ON t.table_name = k3.table_name AND
                        t.constraint_name = k4.constraint_name
                        AND k4.ordinal_position = 4
WHERE t.constraint_type IN ('PRIMARY KEY', 'UNIQUE')
ORDER BY 1
```

Rămâne de încercat pe cont propriu redactarea interogării prin care se reconstruiesc expresiile cheilor străine, folosindu-se, pe lângă tabelele virtuale-dicţionar de până acum, „clasicele" *sys.foreign_keys* şi *sys.foreign_key_columns*, sau „standardizata" REFERENTIAL_CONSTRAINTS din INFORMATION_SCHEMA.

Capitolul 16
Funcții, proceduri și pachete stocate

Dacă procedura sau funcția reprezintă noțiuni vehiculate de multe decenii de informaticienii cu sau fără acte în regulă, asocierea acestora cu persistența (bazele de date) a fost consacrată de SGBD-urile deceniului al nouălea din secolul precedent. Stocarea se referă la „înăuntrul" bazei, la *dicționarul de date* sau *catalogul sistemului*, cum i se spune la case (de soft) mai mari. Deși puternic impregnate cu SQL, funcțiile și procedurile stocate sunt, după cum bine le zice numele, cam... procedurale, depinzând într-o măsură decisivă de extensiile proprietare ale fiecărui SGBD. Nici termenul *procedură* nu e tocmai în regulă, deoarece partea se regăsește în întreg, adică *procedurile stocate* sunt, în general, de trei tipuri: funcții, *proceduri* și declanșatoare. Mai nimerit ar fi termenul *module stocate*, dar dacă nu am fost prea pedanți până în acest capitol, ce rost mai are acum?

În acest capitol ne ocupăm de funcții, proceduri și pachete (pachete în sens *oraclist*), iar în capitolul următor vom trata declanșatoarele (*trigger*-ele). Cu acest capitol ieșim mult de sub incidența standardelor SQL și intrăm pe proprietățile serverelor de baze de date. De peste zece ani, standardul SQL are o componentă specială - *Persistent Stored Modules* - care încearcă să fie un numitor comun al dialectelor procedurale ale serverelor BD. Procesul de armonizare este vizibil, însă diferențele dintre sistemele de gestiune a bazelor de date sunt încă semnificative, uneori uriașe.

Nu mi-am propus, și dealtfel nici nu mă pricep, să tratez *in extenso* subiectul proceduralității în cele patru servere. Însă următoarele câteva exemple pot constitui un bun punct de plecare pentru cei care vor să facă performanță. Ca lucrări de referință v-aș recomanda, printre altele:

- pentru DB2 - limbajul SQL PL: [Mullins 2004], [Janmohamed *et al.* 2004], [Birchall 2007], [Bedoya *et al.* 2004];
- pentru Oracle - limbajul PL/SQL: [Lakshman 2003], [Greenwald *et al.* 2005], [Price 2008], [Kyte 2005], [Feuerstein 2007], [Urman *et al.* 2004], [McLaughlin 2008];
- pentru PostgreSQL - limbajul PL/pgSQL: [Geschwinde & Schönig 2001], [Matthew & Stones 2005], [Douglas & Douglas 2005];
- pentru MS SQL Server - limbajul T-SQL: [Bain *et al.* 2003], [Watt 2007], [Šunderi 2006], [Ben-Gan *et al.* 2006], [Vieira 2007].

16.1. Funcții stocate

Funcțiile stocate reprezintă un ingredient al tuturor SGBD-urilor semnificative de astăzi. De obicei, în programare, se spune că o funcție este un bloc/program care, la invocare, returnează (furnizează) o valoare. Valoarea poate fi una simplă, de tip înregistrare sau seturi de înregistrări. Pentru variație, modificăm ordinea de intrare în scenă a celor patru servere BD în abordarea subiectului. Vom începe cu PostgreSQL-ul, vom continua cu Oracle şi SQL Server şi vom încheia cu DB2.

16.1.1. Primele funcții stocate în PostgreSQL

În PostgreSQL, funcțiile stocate pot fi create în mai multe limbaje: Perl, C etc. Pe noi ne interesează doar două dintre acestea, SQL şi PLpg/SQL. Cele mai simple sunt cele SQL. De obicei, acestea sunt simple interogări parametrizate care returnează o valoare. Spre exemplu, în listingul 16.1 apar două comenzi (CREATE FUNCTION) prin care se creează funcții. Prima, denumită *f_proctva*, primeşte, la invocare (apelare) valoarea unui cod de produs şi furnizează procentul său de TVA. A doua, *f_denpr*, primeşte un cod de produs şi furnizează denumirea acestuia.

Listingul 16.1. Primele două funcții stocate PostgreSQL, redactate în limbajul SQL

```
-- F_PROCTVA -- pentru un cod de produs returneaza procentul de TVA al produsului respectiv
CREATE OR REPLACE FUNCTION f_proctva (codpr_ NUMERIC)RETURNS NUMERIC AS $$
    SELECT proctva FROM produse WHERE codpr = $1
$$ LANGUAGE SQL;

-- F_DENPR - primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr (codpr_ produse.codpr%TYPE)
    RETURNS produse.denpr%TYPE AS $$
    SELECT denpr FROM produse WHERE codpr = $1;
$$ LANGUAGE SQL;
```

Ambele funcții folosesc ca parametru de intrare *CodPr_* care este definit direct, *NUMERIC*, în prima funcție şi indirect - *PRODUSE.CodPr%TYPE* -, în a doua. Ultimul mod de specificare a tipului unui parametru este poate mai lung, însă mai elegant. Dacă, ulterior, s-ar modifica tipul atributului CodPr din PRODUSE (din NUMERIC în VARCHAR, nu mă întrebați de ce), ar trebui modificate toate funcțiile stocate care prezintă parametri de genul *CodPr_*, definiți ca NUMERIC[1], în timp ce funcțiile în care parametrii au fost definiți indirect nu suferă modificări (eventual, ar trebui recompilate). De remarcat că, în corpul funcției, referirea la parametrul *codpr_* se realizează prin *$1* (dacă funcția ar fi avut trei parametri, referirea lor s-ar fi făcut prin *$2* şi *$3*)[2].

Lansarea în execuție a celor două comenzi CREATE FUNCTION poate fi realizată precum în figura 16.1, deşi o variantă mai simplă ar fi lucrul asistat (chiar dacă în *pgAdmin* lucrul asistat nu este întotdeauna o bagatelă).

1. E un fel de miniproblemă a anului 2000 (Y2K) de care cei mai în vârstă dintre dumneavoastră ați auzit în tinerețe.
2. La drept vorbind, numele parametrului poate lipsi, doar tipul său fiind important.

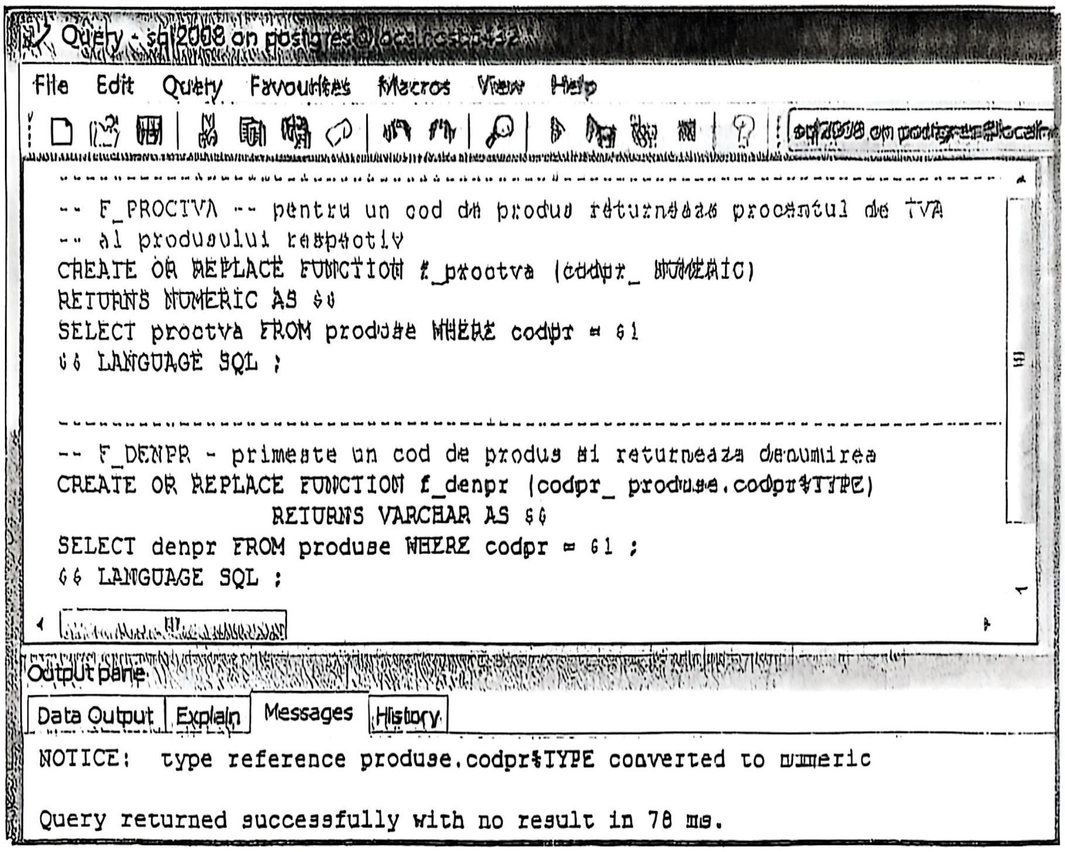


Figura 16.1. Crearea a două funcții (SQL) simple (de tot) în PostgreSQL

După lansarea acestor comenzi, în *pgAdmin*, cele două funcții pot fi vizualizate ca în figura 16.2 (nu uitați de reîmprospătarea - *Refresh* - meniului arborescent). Odată creată, o funcție poate fi apelată atât din altă funcție, cât și dintr-o interogare SQL:

```
SELECT nrfact, f_denpr(codpr) AS denpr, cantitate, pretunit,
        cantitate * pretunit AS Val_fara_tva,
        cantitate * pretunit * f_proctva(codpr) AS tvalinie,
        cantitate * pretunit * (1 + f_proctva(codpr)) AS Val_cu_TVA_linie
FROM liniifact
```

În PostgreSQL, o funcție SQL poate conține mai multe comenzi SQL, însă valoarea, linia sau setul de linii sunt returnate pe baza ultimei comenzi SQL. De asemenea, dacă valoarea returnată declarată prin clauza RETURNS este una simplă (o valoare sau o înregistrare), iar ultimul SELECT al funcției extrage două sau mai multe întregistrări dintr-o tabelă, nu se va declanșa vreo eroare (precum în Oracle PL/SQL), ci se va returna prima linie din setul de înregistrări, ceea ce poate crea confuzii (mai ales în lipsa opțiunii ORDER BY în această ultimă interogare a funcției).

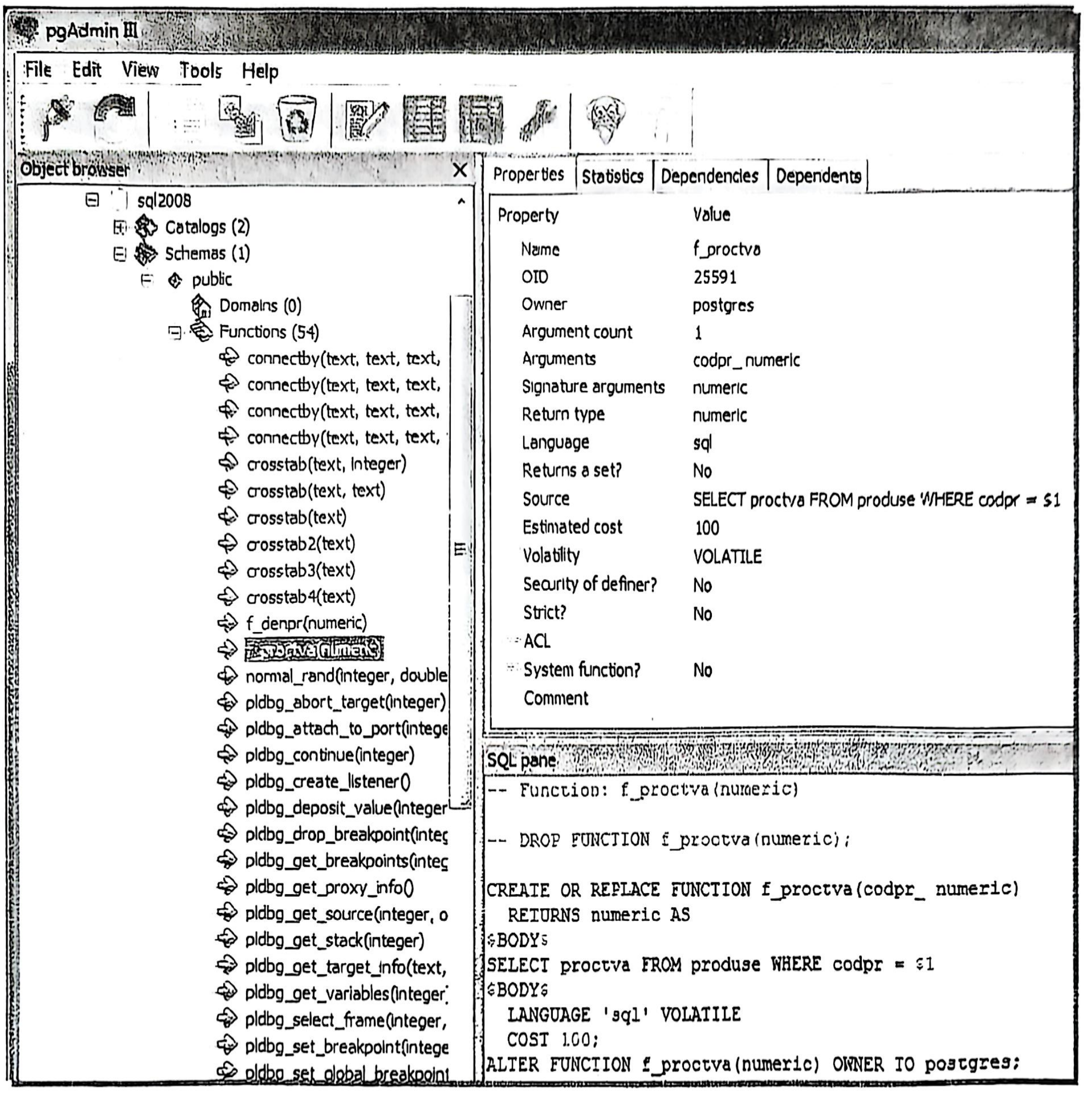


Figura 16.2. Afişarea celor două funcţii în pgAdmin

Să luăm, spre exemplu, funcţia *f_test01* (listingul 16.2) alcătuită din trei fraze SELECT destul de anapoda, care extrag, pe rând, înregistrările din tabelele CLIENŢI, PERSOANE şi FACTURI (ultima ar extrage numai valorile atributului NrFact din toate liniile tabelei FACTURI):

Listingul 16.2. Funcţia SQL (din PostgreSQL) ***f_test01***

```
CREATE OR REPLACE FUNCTION f_test01 () RETURNS NUMERIC AS $$
    SELECT * FROM clienti;
    SELECT * FROM persoane;
    SELECT NrFact FROM facturi;
$$ LANGUAGE SQL;
```

E oarecum curios că sintaxa aceastei funcţii este acceptată în PostgreSQL. Întrucât valoarea declarată în clauza RETURNS este simplă (numerică), apelul funcţiei nu se soldează cu eroare, ci cu vizualizarea unui număr de factură (vezi figura 16.3).

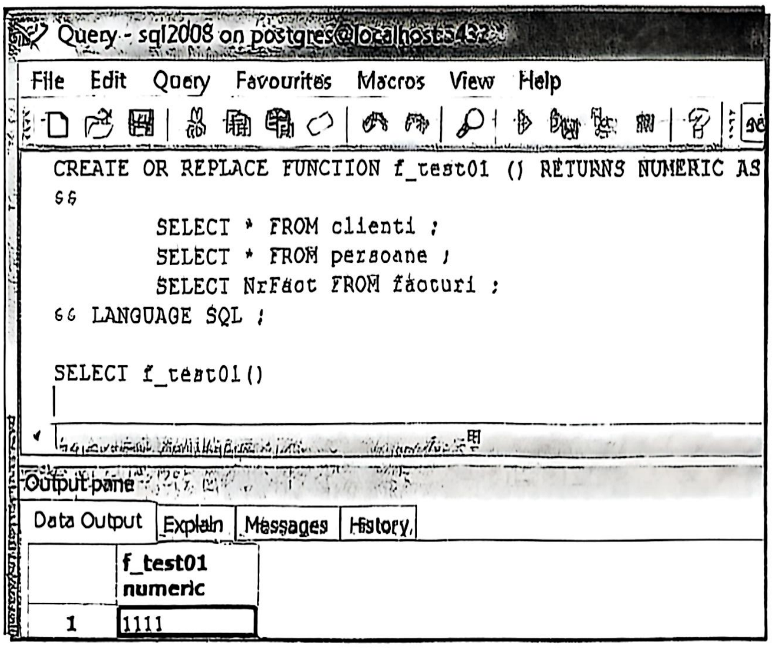


Figura 16.3. O funcție Postgre(SQL) ciudată și rezultatul său

Limbajul „clasic" de redactare a funcțiilor stocate complexe este, în PostgreSQL, PL/pgSQL. Sintaxa sa este similară, în multe privințe, limbajului Oracle PL/SQL. Reluăm cele două funcții anterioare și le rescriem pe sintaxa PL/pgSQL (vezi listingul 16.3).

Listingul 16.3. Primele două funcții „trecute" în PL/pgSQL

```
-- F_PROCTVA -- pentru un cod de produs returneaza procentul de TVA al produsului
CREATE OR REPLACE FUNCTION f_proctva2 (codpr_ produse.codpr%TYPE)
    RETURNS produse.ProcTVA%TYPE AS $$
DECLARE
    v_proc produse.proctva%TYPE;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_;
    RETURN v_proc;
END;
$$ LANGUAGE plpgsql;
--------------------------------------------------------------------
-- F_DENPR - primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr2 (codpr_ produse.codpr%TYPE)
    RETURNS produse.denpr%TYPE AS $$
DECLARE
    v_denpr produse.denpr%TYPE;
BEGIN
    SELECT denpr INTO v_denpr FROM produse WHERE codpr = codpr_;
    RETURN v_denpr;
END;
$$ LANGUAGE plpgsql;
```

Un bloc (program) scris în limbajul PL/pgSQL poate avea trei secțiuni: prima (opțională) este declarativă - începe cu DECLARE; a doua este cea „executabilă" - începe cu BEGIN; a treia este tot opțională și este destinată tratării eventualelor erori ce pot fi declanșate de funcție, erori denumite *excepții*. Spre deosebire de

funcţiile redactate în „limbajul" SQL, în PL/pgSQL, parametrii sunt referiţi prin numele lor.

Ca şi în alte limbaje, putem să redactăm funcţii care să suplinească oferta sistemului. Spre exemplu, în PostgreSQL nu există funcţiile LEFT şi RIGHT (din SQL Server), care furnizează primele, respectiv ultimele *n* caractere dintr-un şir. Aşa că putem redacta noi două funcţii pe care le vom numi LEFT_ şi RIGHT_ (am adăugat „un underscore" tocmai pentru a nu intra în conflict cu o eventuală viitoare versiune PostgreSQL care va implementa cele două funcţii. Iată aceste două funcţii în listingul 16.4.

Listingul 16.4. Funcţiile ***left_*** şi ***right_*** (în PL/pgSQL)

```
-- Functia LEFT_ extrage primele cate_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION LEFT_ (sir_ TEXT, cate_ INTEGER) RETURNS TEXT
AS $$
DECLARE
     v_sir_returnat TEXT;
BEGIN
     v_sir_returnat := SUBSTR(sir_, 1, cate_);
     RETURN v_sir_returnat;
END;
$$ LANGUAGE plpgsql;

-- Functia RIGHT_ extrage ultimele cate_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION RIGHT_ (sir_ TEXT, cate_ INTEGER) RETURNS TEXT
AS $$
DECLARE
     v_sir_returnat TEXT;
BEGIN
     v_sir_returnat := SUBSTR(sir_, LENGTH(sir_) - cate_ + 1, cate_);
     RETURN v_sir_returnat;
END;
$$ LANGUAGE plpgsql;
```

Ilustrarea secvenţelor alternative şi iterative în PL/pgSQL ne „împinge" la redactarea unei funcţii care să rezolve o problemă destul de importantă – calcularea scadenţei facturilor peste un număr de zile *lucrătoare*. Funcţia de numeşte *f_data_scadenta* (vezi listingul 16.5).

Listingul 16.5. Funcţia PLpgSQL ***f_data_scadenta***

```
CREATE OR REPLACE FUNCTION f_data_scadenta (
          datai_ DATE, nrzile_scadenta_ NUMERIC) RETURNS DATE AS $$
DECLARE
     zi_crt DATE;
     v_gata BOOLEAN := FALSE;
     v_nr_zile_lucratoare NUMERIC := 0;
BEGIN
     -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
     zi_crt := datai_ ;
     WHILE NOT v_gata LOOP
          zi_crt := zi_crt + 1;
          IF RTRIM(TO_CHAR(zi_crt, 'DAY')) NOT IN ('SATURDAY','SUNDAY') THEN
                    v_nr_zile_lucratoare := v_nr_zile_lucratoare + 1;
                    IF v_nr_zile_lucratoare >= nrzile_scadenta_ THEN
                              EXIT;
```

```
                END IF;
            END IF;
        END LOOP;
        RETURN zi_crt;
    END;
    $$ LANGUAGE plpgsql;
```

Dacă presupunem că scadenţa fiecărei facturi este peste 19 zile lucrătoare de la data întocmirii, atunci interogarea devine foarte simplă:

```
SELECT NrFact, DataFact,
f_data_scadenta (DataFact, 19) AS "Scadenta (19 zile lucratoare)"
FROM facturi
```

iar rezultatul este cel din figura 16.4.

nrfact numeric(8,0)	datafact date	Scadenta (19 zile lucratoare) date
1111	2007-08-01	2007-08-28
1112	2007-08-01	2007-08-28
1113	2007-08-01	2007-08-28
1114	2007-08-01	2007-08-28
1115	2007-08-02	2007-08-29
1116	2007-08-02	2007-08-29
1117	2007-08-03	2007-08-30
1118	2007-08-04	2007-08-30
1119	2007-08-07	2007-09-03
1120	2007-08-07	2007-09-03
1121	2007-08-07	2007-09-03
2111	2007-08-14	2007-09-10

Figura 16.4. Scadenţă peste 19 zile lucrătoare

Acum ne propunem să afişăm în dreptul fiecărei facturi, sub formă de şir de caractere lista produselor (denumirile lor) care o alcătuiesc, ca în figura 16.5. O variantă de rezolvare a problemei se bazează pe două funcţii numite *f_codpr_liniefact* şi *f_produse_din_factura_v1* (vezi listingul 16.6). Prima primeşte doi parametri, un număr de factură (*nrfact_*) şi un număr de linie (*linie_*) şi returnează codul produsului de pe linia/factura respectivă. În cazul în care nu există linia cu numărul *linie_* în factura *nrfact_*, funcţia returnează valoarea 0. A doua - *f_produse_din_factura_v1* - construieşte şirul ce conţine denumirile produselor din factura *nrfact_* printr-o buclă executată de maximum 99 ori, aceasta deoarece presupunem că o factură conţine cel mult 99 de linii (ceea ce e destul de improbabil). Cum liniile din toate facturile sunt dispuse consecutiv, în momentul în care se depistează o linie inexistentă, se iese forţat (EXIT) din buclă, aşa că nu trebuie să ne facem griji pentru mersul în gol al iteraţiei.

Listingul 16.6. Funcţiile ***f_codpr_liniefact*** şi ***f_produse_din_factura_v1***

```
CREATE OR REPLACE FUNCTION f_codpr_liniefact
        (nrfact_ liniifact.nrfact%TYPE, linie_ liniifact.linie%TYPE)
    RETURNS liniifact.codpr%TYPE AS $$
DECLARE
```

```
    v_codpr liniifact.codpr%TYPE := 0;
BEGIN
    SELECT CodPr INTO v_codpr FROM liniifact WHERE NrFact = nrfact_ AND Linie = linie_;
    RETURN v_codpr;
END;
$$ LANGUAGE plpgsql;
-------------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION f_produse_din_factura_v1
    (nrfact_ facturi.nrfact%TYPE) RETURNS TEXT AS $$
DECLARE
    v_sir TEXT := '';
BEGIN
    FOR i IN 1..99 LOOP
        IF f_codpr_liniefact (CAST(nrfact_ AS INTEGER), CAST( i AS SMALLINT)) > 0 THEN
            v_sir := v_sir || ' * ' ||
            f_denpr(f_codpr_liniefact (CAST (nrfact_ AS INTEGER), CAST (i AS SMALLINT)) );
        ELSE
            EXIT;
        END IF;
    END LOOP;
    RETURN v_sir;
END;
$$ LANGUAGE plpgsql;
```

Pentru a preîntâmpina erorile legate de nepotrivirile de tip între parametrii specificaţi la apelarea funcţiei *f_codpr_liniefact* şi cei declaraţi în corpul funcţiei respective, folosim funcţia CAST. Extragerea denumirii unui produs pe baza codului său ne prilejuieşte folosirea funcţiei discutate la începuturile paragrafului, *f_denpr*. Raportul din figura 16.5 este obţinut acum prin interogarea:

SELECT facturi.NrFact, facturi.DataFact,

f_produse_din_factura_v1(facturi.NrFact) AS Lista_produse

FROM facturi ORDER BY 1

nrfact integer	datafact date	lista_produse text
1111	2007-08-01	* Produs 1 * Produs 2 * Produs 5
1112	2007-08-01	* Produs 2 * Produs 3
1113	2007-08-01	* Produs 2
1114	2007-08-01	* Produs 2 * Produs 4 * Produs 5
1115	2007-08-02	* Produs 2
1116	2007-08-02	* Produs 2
1117	2007-08-03	* Produs 2 * Produs 1
1118	2007-08-04	* Produs 2 * Produs 1
1119	2007-08-07	* Produs 2 * Produs 3 * Produs 4 * Produs 5
1120	2007-08-07	* Produs 2
1121	2007-08-07	* Produs 5 * Produs 2
1122	2007-08-07	
2111	2007-08-14	* Produs 1 * Produs 2 * Produs 5
2112	2007-08-14	* Produs 2 * Produs 3

Figura 16.5. Produsele din componenţa fiecărei facturi

16.1.2. Primele funcții stocate în Oracle PL/SQL

De ani buni, există posibilitatea redactării procedurilor stocate Oracle în limbajul Java, însă limbajul predilect de redactare a funcțiilor (procedurilor, pachetelor și declanşatoarelor) rămâne PL/SQL. Nu vom intra în detalii privind „filosofia limbajului”[3], ci doar vom prezenta câteva dintre opțiunile de lucru. Ca și PL/pgSQL (pe care l-a inspirat copios), PL/SQL-ul pune la dispoziția programatorilor trei secțiuni, declarativă, executabilă și de tratare a excepțiilor. Să începem cu redactarea funcțiilor *f_proctva* și *f_denpr* (vezi listingul 16.7).

Listingul 16.7. Funcțiile PL/SQL ***f_proctva*** și ***f_denpr***

```
-- F_PROCTVA pentru un cod de produs returneaza procentul de TVA al produsului
CREATE OR REPLACE FUNCTION f_proctva2 (codpr_ produse.codpr%TYPE)
    RETURN produse.ProcTVA%TYPE
AS
    v_proc produse.proctva%TYPE;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_;
    RETURN v_proc;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN NULL;
END;
/
----------------------------------------------------------------
-- F_DENPR primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr2 (codpr_ produse.codpr%TYPE)
    RETURN produse.denpr%TYPE
AS
    v_denpr produse.denpr%TYPE;
BEGIN
    SELECT denpr INTO v_denpr FROM produse WHERE codpr = codpr_;
    RETURN v_denpr;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN NULL;
END;
/
```

Câteva deosebiri față de versiunea PL/pgSQL trebuie subliniate. În procedurile și funcțiile PL/SQL, secțiunea declarativă nu începe cu DECLARE, ci urmează clauzei AS sau IS. Interogarea SELECT trebuie să extragă din tabela/tabelele din clauza FROM nici mai mult nici mai puțin de o linie, altminteri ne vom procopsi cu un mesaj de eroare - NO_DATA_FOUND. Orice eroare (excepție) este tratată în secțiunea excepțiilor. Atunci când valoarea parametrului de intrare nu este în tabela PRODUSE, în locul blocării eventualei aplicații, se returnează NULL.

3. Pentru o tratare mai detaliată a limbajului, vezi și [Fotache *et al.* 2003].

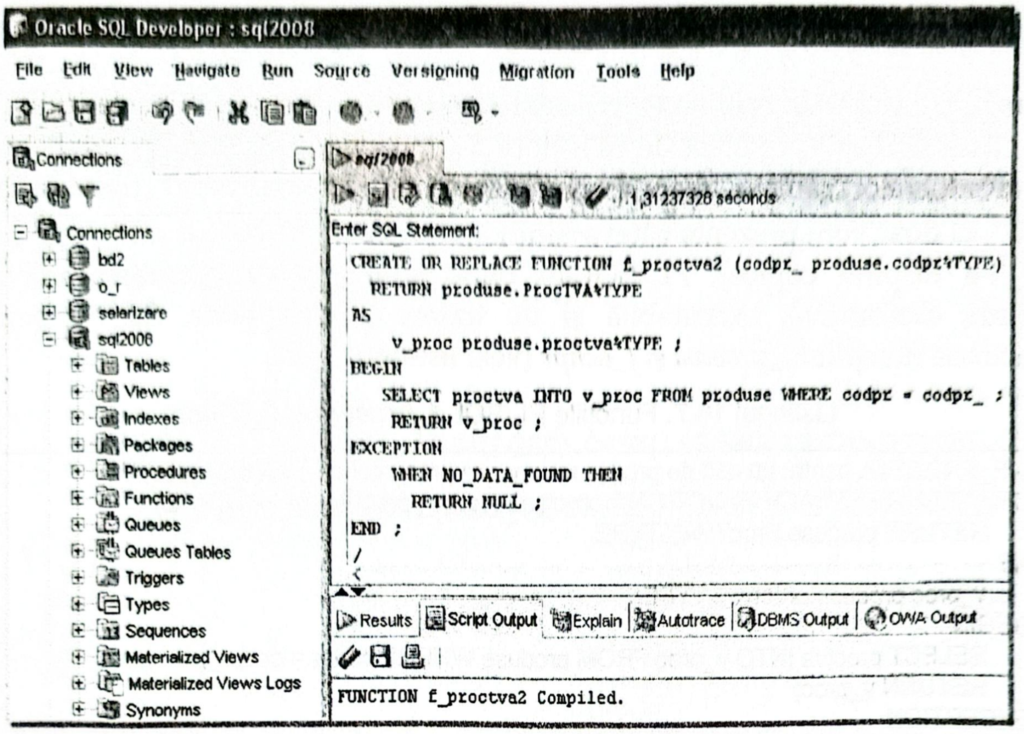


Figura 16.6. Crearea directă a unei funcţii în Oracle SQL Developer (prin comanda CREATE FUNCTION)

Mai trebuie spus că tipurile definite pentru variabile în blocurile PL/SQL sunt mai generoase decât cele discutate în capitolul 3. Astfel, într-o tabelă un atribut nu poate fi declarat de tip BOOLEAN (logic); în schimb, orice variabilă poate fi declarată astfel. La fel stau lucrurile cu tipurile *integer* – BINARY_INTEGER sau PLS_INTEGER –, cu vectorii asociativi etc.

O primă modalitate de creare a funcţiilor în SQL Developer este scriere directă sau copierea dintr-un alt fişier (text), ca în figura 16.6, comanda fiind, fireşte, CREATE OR REPLACE FUNCTION... A doua modalitate este cea asistată (vezi figura 16.7). În meniul arborescent, în cadrul schemei curente ne deplasăm pe nodul *Functions* şi apelăm meniul contextual, care ne va afişa şapte opţiuni (stânga figurii) din care alegem *New Function*. Urmează să definim numele funcţiei, parametrul (parametrii) şi tipul datelor returnate, iar apoi să completăm definiţia funcţiei. În final, compilăm funcţia pentru a vedea ce erori s-au strecurat.

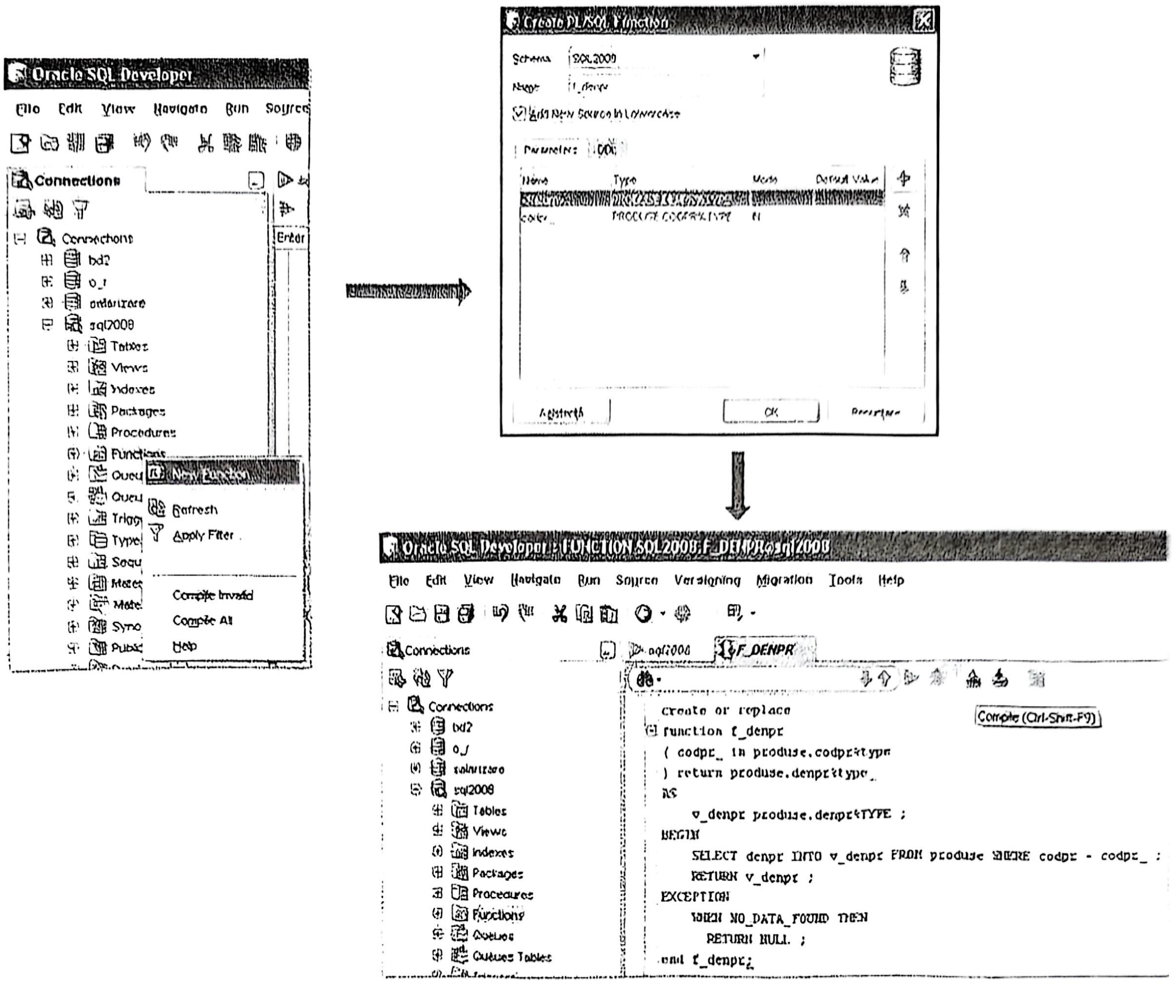


Figura 16.7. Crearea asistată a unei funcții în Oracle SQL Developer

Urmând exemplele din paragraful anterior, creăm în PL/SQL cele două funcții care suplinesc lipsa funcțiilor LEFT și RIGHT (existente în SQL Server), funcții care se află în listingul 16.8.

Listingul 16.8. Funcțiile (PL/SQL) ***left_*** și ***right_***

```
-- Functia LEFT_ extrage primele cate_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION LEFT_ (sir_ VARCHAR2, cate_ NUMERIC)
    RETURN VARCHAR2
AS
    v_sir_returnat VARCHAR2(5000);
BEGIN
    v_sir_returnat := SUBSTR(sir_, 1, cate_);
    RETURN v_sir_returnat;
END LEFT_;
/
-- Functia RIGHT_ extrage ultimele cate_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION RIGHT_ (sir_ VARCHAR2, cate_ NUMERIC)
    RETURN VARCHAR2
AS
    v_sir_returnat VARCHAR2(5000);
BEGIN
    v_sir_returnat := SUBSTR(sir_, LENGTH(sir_) - cate_ + 1, cate_);
    RETURN v_sir_returnat;
END;
/
```

În continuare (listingul 16.9), portăm funcţia *f_data_scadenta* (pentru determinarea scadenţei facturilor peste un număr de zile lucrătoare) din PL/pgSQL în PL/SQL. După cum se poate observa, deosebirile sunt minore.

Listingul 16.9. Funcţia (PL/SQL) ***f_data_scadenta***

```
CREATE OR REPLACE FUNCTION f_data_scadenta (
  datai_ DATE, nrzile_scadenta_ NUMERIC) RETURN DATE
AS
    zi_crt DATE;
    v_gata BOOLEAN := FALSE;
    v_nr_zile_lucratoare NUMERIC := 0;
BEGIN
    -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
    zi_crt := datai_;
    WHILE NOT v_gata LOOP
        zi_crt := zi_crt + 1;
        IF RTRIM(TO_CHAR(zi_crt, 'DAY')) NOT IN ('SATURDAY','SUNDAY') THEN
                v_nr_zile_lucratoare := v_nr_zile_lucratoare + 1;
                IF v_nr_zile_lucratoare >= nrzile_scadenta_ THEN
                        EXIT;
                END IF;
        END IF;
    END LOOP;
    RETURN zi_crt;
END;
/
```

Portăm şi cele două funcţii PL/pgSQL din listingul 16.6, varianta „oraclistă" fiind cea din listingul 16.10. În prima, atunci când nu există linia din factura specificată, funcţia va returna NULL.

Listingul 16.10. Funcţiile PL/SQL ***f_codpr_liniefact*** şi ***f_produse_din_factura_v1***

```
CREATE OR REPLACE FUNCTION f_codpr_liniefact
        (nrfact_ liniifact.nrfact%TYPE, linie_ liniifact.linie%TYPE)
    RETURN liniifact.codpr%TYPE AS
    v_codpr liniifact.codpr%TYPE := 0;
BEGIN
    SELECT CodPr INTO v_codpr FROM liniifact WHERE NrFact=nrfact_ AND Linie=linie_;
    RETURN v_codpr;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  RETURN NULL;
END f_codpr_liniefact;
/
--------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION f_produse_din_factura_v1
    (nrfact_ facturi.nrfact%TYPE) RETURN VARCHAR2
AS
  v_sir VARCHAR2(1000) := '';
BEGIN
    FOR i IN 1..99 LOOP
        IF f_codpr_liniefact (nrfact_ , i) > 0 THEN
                v_sir := v_sir || ' * ' || f_denpr(f_codpr_liniefact (nrfact_, i) );
        ELSE
                EXIT;
        END IF;
    END LOOP;
```

```
        RETURN v_sir;
    END f_produse_din_factura_v1;
```

16.1.3. Primele funcții stocate în SQL Server

Limbajul de programare al SQL Server este Transact-SQL, pe scurt T-SQL. Sintaxa sa diferă serios de cea a PL/pgSQL şi PL/SQL, însă vom putea formula soluții echivalente fără un efort prea mare. Iată în listingul 16.11 comanda de creare a funcției *f_proctva* în schema curentă (dbo). Comanda de creare a funcției este precedată de un IF care şterge funcția din dicționar, în caz că aceasta există. IF-ul este util, pentru că T-SQL nu acceptă varianta CREATE OR REPLACE FUNCTION...

Listingul 16.11. Funcția (T-SQL) ***f_proctva***

```
IF OBJECT_ID (N'dbo.f_proctva', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_proctva;
GO
CREATE FUNCTION f_proctva (@codpr SMALLINT)
     RETURNS NUMERIC(2,2)
WITH EXECUTE AS CALLER
AS
BEGIN
  DECLARE @v_proc NUMERIC(2,2)
     SELECT @v_proc = ProcTVA FROM produse WHERE CodPr = @codpr
  RETURN(@v_proc)
END;
```

Nu există o secțiune declarativă specială, şi niciun caracter separator între comenzi. De asemenea, variabilele (ale căror nume este prefixat de @) se declară numai direct, iar formatul de atribuire a valorii unei variabile pe baza unei interogări este unul particular T-SQL. Funcția *f_denpr* (listingul 16.12) este similară.

Listingul 16.12. Funcția (T-SQL) ***f_denpr***

```
IF OBJECT_ID (N'dbo.f_denpr', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_denpr;
GO
CREATE FUNCTION f_denpr (@codpr SMALLINT)
     RETURNS VARCHAR(30)
WITH EXECUTE AS CALLER
AS
BEGIN
  DECLARE @v_denpr VARCHAR(30)
     SELECT @v_denpr = DenPr FROM produse WHERE CodPr = @codpr
  RETURN(@v_denpr)
END;
```

Ca şi în precedentele servere, o funcție poate fi creată fie direct, fie interactiv. Varianta directă nu diferă de suratele sale din pgAdmin şi SQL Developer (vezi figura 16.8).

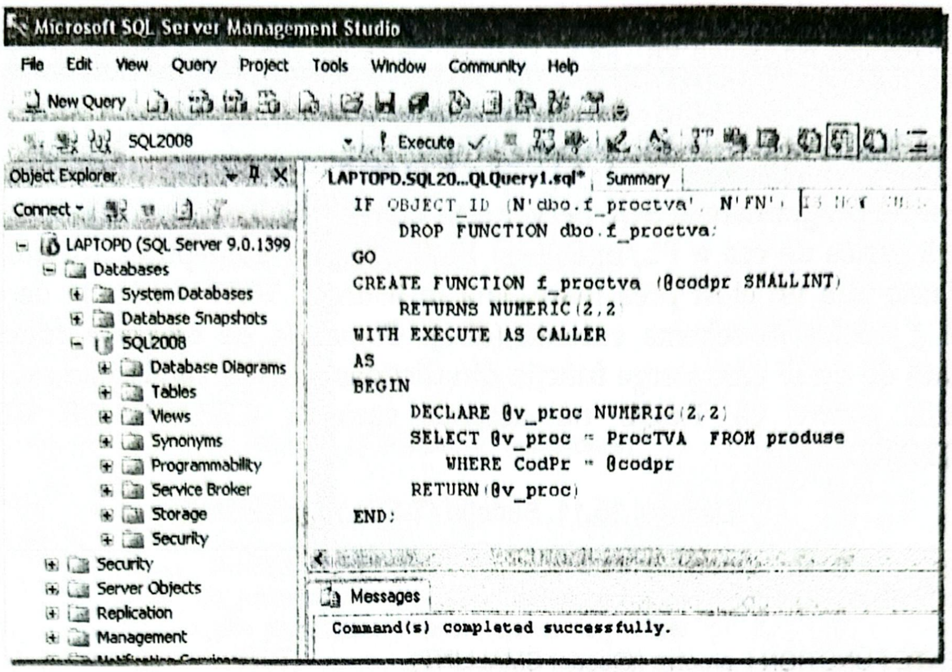


Figura 16.8. Crearea directă a unei funcţii T-SQL

Varianta asistată este mult mai spectaculoasă întrucât *SQL Server Management Studio* este extrem de generos în afişarea unui model (*template*) în care sunt incluse cele mai frecvente comenzi/instrucţiuni (vezi figura 16.9).

O altă deosebire faţă de PL/pgSQL şi PL/SQL este că în T-SQL nu există tipul BOOLEAN. Putem folosi un tip la care nu am apelat până acum, şi anume BIT. Tipul BIT permite numai două valori, 0 şi 1, plus NULL. Variabila *@v_gata* din funcţia *f_data_scadenta* (listingul 16.13) este de acest tip.

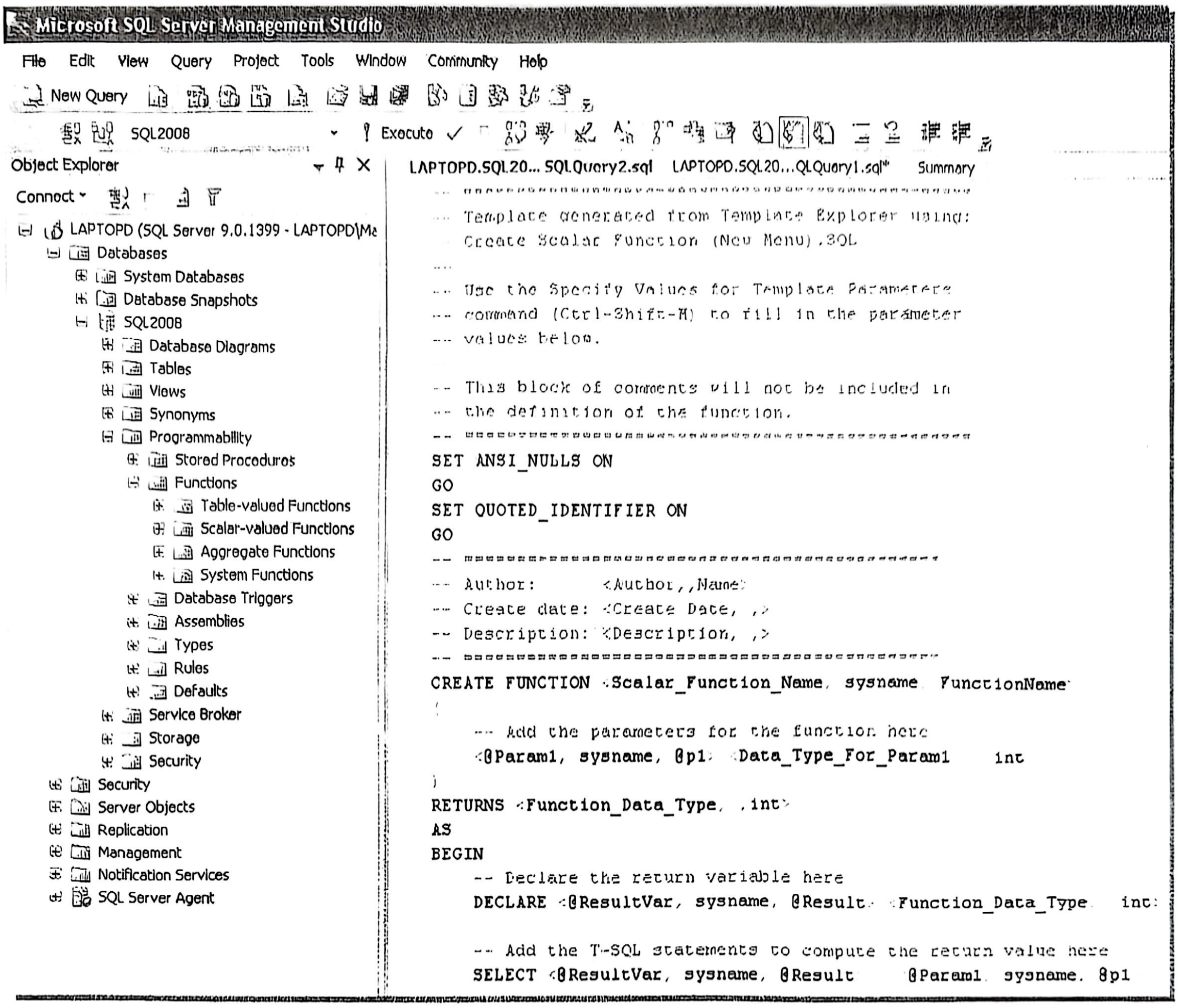


Figura 16.9. Crearea asistată a unei funcții scalare în T-SQL

Comanda de atribuire este SET, iar pentru un mai bun control al blocurilor executate pe ramurile THEN și ELSE ale unei structuri IF sau CASE sau al blocurilor din corpul unei bucle (structuri iterative), se folosesc „marcajele" BEGIN și END. Tot în acest listing observăm că EXIT-ului din PL/(pg)SQL îi corespunde BREAK în T-SQL.

Listingul 16.13. Funcția (T-SQL) ***f_data_scadenta***

```
IF OBJECT_ID (N'dbo.f_data_scadenta', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_data_scadenta;
GO
CREATE FUNCTION f_data_scadenta (@datai_ SMALLDATETIME, @nrzile_scadenta_ TINYINT)
    RETURNS SMALLDATETIME AS
BEGIN
    DECLARE @zi_crt SMALLDATETIME;
    DECLARE @v_gata BIT;
    DECLARE @v_nr_zile_lucratoare TINYINT;
    SET @v_gata = 0;
    SET @v_nr_zile_lucratoare = 0;

    -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
    SET @zi_crt = @datai_;
    WHILE @v_gata = 0 BEGIN
        SET @zi_crt = DATEADD(DAY, 1, @zi_crt);
```

```
            IF DATEPART(dw,@zi_crt) NOT IN (1,7)
                    SET @v_nr_zile_lucratoare = @v_nr_zile_lucratoare + 1;
                    IF @v_nr_zile_lucratoare >= @nrzile_scadenta_
                              BREAK;
     END;
     RETURN @zi_crt;
END;
```

Invocarea funcţiei într-o interogare nu ridică nicio problemă:

SELECT NrFact, DataFact,

dbo.f_data_scadenta (DataFact, 19) AS "Scadenta (19 zile lucratoare)"

FROM facturi

În SQL Server, există funcţiile LEFT şi RIGHT, aşa că nu mai are rost să creăm funcţii echivalente, dar nu există INITCAP, aşa că avem ocupaţie (vezi listingul 16.14). În afara perechii BEGIN-END corespunzătoare blocului principal, mai avem una ce delimitează secvenţa iterativă (WHILE...) şi o altă pereche ce marchează ramura THEN a testului *IF SUBSTRING...*

Listingul 16.14. Funcţia (T-SQL) ***INITCAP_***

```
IF OBJECT_ID (N'dbo.INITCAP_', N'FN') IS NOT NULL
  DROP FUNCTION dbo.INITCAP_;
GO
CREATE FUNCTION dbo.INITCAP_ (@textsursa VARCHAR(100))
    RETURNS VARCHAR (100)
AS
BEGIN
    DECLARE @text_returnat VARCHAR(100);
    DECLARE @i_inceput_cuvant TINYINT;
    DECLARE @i TINYINT;
    SET @text_returnat = @textsursa;
    SET @i = 2;
    SET @i_inceput_cuvant = 1;
    WHILE @i <= LEN(@textsursa)
        BEGIN
        IF SUBSTRING(@textsursa, @i, 1) IN (' ', '-', '.') OR @i=LEN(@textsursa)
                BEGIN
                SET @text_returnat =
                        CASE WHEN @i_inceput_cuvant = 1 THEN '' ELSE
                                SUBSTRING(@text_returnat, 1, @i_inceput_cuvant - 1) END
                                +
                        UPPER(SUBSTRING(@text_returnat, @i_inceput_cuvant, 1)) +
                        SUBSTRING(@text_returnat, @i_inceput_cuvant + 1, LEN(@textsursa) -
                                @i_inceput_cuvant + 1);
                SET @i_inceput_cuvant = @i + 1;
                END;
        SET @i = @i + 1;
        END;

    RETURN @text_returnat;
END;
```

Testarea funcţiei se poate face atât cu un şir oarecare:

SELECT dbo.INITCAP_('ana are mere')

cât şi cu un atribut dintr-o tabelă:

SELECT c.DenCl, c.Adresa, dbo.INITCAP_(c.Adresa) FROM clienti c

Încheiem cu cele două funcţii care furnizează pentru fiecare factură un şir de caractere ce conţine denumirile produselor din factura respectivă (vezi listingul 16.15).

Listingul 16.15. Funcţiile (T-SQL) *f_codpr_liniefact* şi *f_produse_din_factura_v1*

```
IF OBJECT_ID (N'dbo.f_codpr_liniefact', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_codpr_liniefact;
GO
CREATE FUNCTION f_codpr_liniefact (@nrfact INT, @linie TINYINT)
    RETURNS SMALLINT
AS
BEGIN
    DECLARE @v_codpr SMALLINT;
    SET @v_codpr = 0;
    SELECT @v_codpr = CodPr FROM liniifact WHERE NrFact=@nrfact AND Linie=@linie;
    RETURN @v_codpr;
END ;
-----------------------------------------------------------------------------------------
IF OBJECT_ID (N'dbo.f_produse_din_factura_v1', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_produse_din_factura_v1;
GO
CREATE FUNCTION f_produse_din_factura_v1
    (@nrfact INT) RETURNS VARCHAR (1000)
AS
BEGIN
    DECLARE @v_sir VARCHAR(1000);
    DECLARE @i TINYINT;
    SET @v_sir = '';
    SET @i = 0;
    WHILE @i <= 99 BEGIN
        SET @i = @i + 1
        IF dbo.f_codpr_liniefact (@nrfact , @i) = 0
                BREAK
        ELSE
                SET @v_sir = @v_sir + ' * ' + dbo.f_denpr(dbo.f_codpr_liniefact (@nrfact, @i) )
    END
    RETURN @v_sir
END
```

16.1.4. Primele funcţii stocate în DB2

Limbajul de redactare a procedurilor stocate în DB2 se numeşte SQL PL. Sintaxa sa este elegantă, însă dificultatea noastră principală ţine de incapacitatea *DB2 Control Center* de a asigura un suport adecvat creării procedurilor şi funcţiilor stocate. Documentaţia DB2 recomandă folosirea pachetului software *IBM Data Studio,* aşa că, din acest paragraf, vom apela la serviciile sale.

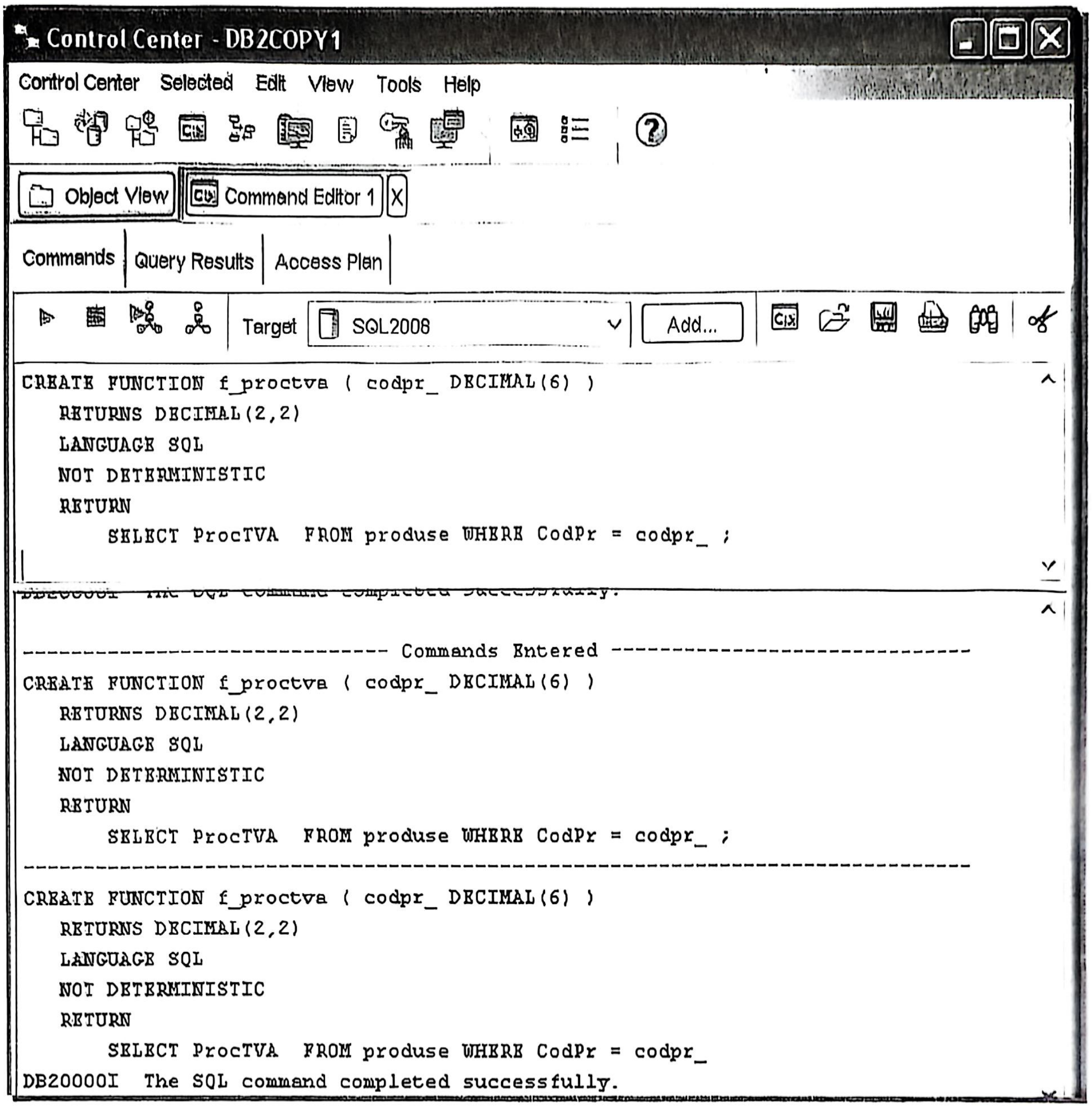


Figura 16.10. Crearea directă a unei funcţii SQL PL în DB2 Control Center

Începem prin a porta în SQL PL primele două funcţii, *f_proctva* şi *f_denpr* (listingul 16.16). După specificaţii, singura comandă a funcţiilor este RETURN care are ca argument o interogare.

Listingul 16.16. Funcţiile SQL PL ***f_proctva*** şi ***f_denpr***

```
CREATE FUNCTION f_proctva ( codpr_ SMALLINT )
    RETURNS DECIMAL(2,2)
    LANGUAGE SQL
    NOT DETERMINISTIC
RETURN
    SELECT ProcTVA FROM produse WHERE CodPr = codpr_;

CREATE FUNCTION f_denpr ( codpr_ SMALLINT )
    RETURNS VARCHAR(30)
    LANGUAGE SQL
    NOT DETERMINISTIC
RETURN
    SELECT DenPr FROM produse WHERE CodPr = codpr_;
```

În DB2 *Control Center*, funcţiile trebuie scrise (sau copiate) şi apoi compilate (vezi figura 16.10). Compilarea decurge fără probleme pentru că funcţiile sunt simple. Figura 16.11 ilustrează modul de creare a uneia dintre cele două funcţii cu ajutorul *IBM Data Studio*.

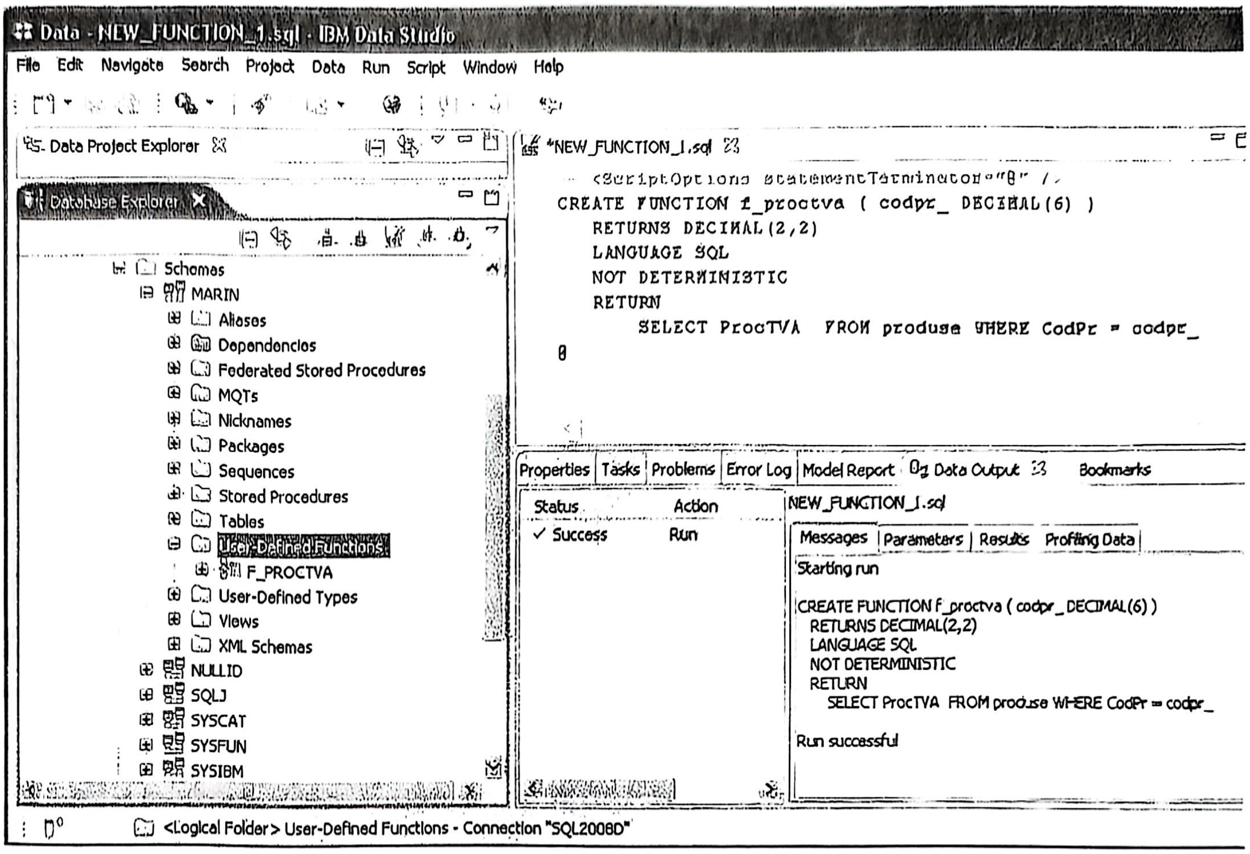


Figura 16.11. Crearea unei funcţii SQL în IBM Data Studio

Nici funcţiile *LEFT_* şi *RIGHT_* (listingul 16.17) nu ridică probleme deosebite şi acestea având o singură comandă – RETURN.

Listingul 16.17. Funcţiile SQL PL ***LEFT_*** şi ***RIGHT_***

```
-- Functia LEFT_ extrage primele cate_ caractere dintr-un sir_
CREATE FUNCTION LEFT_ (sir_ VARCHAR(1000), cate_ INTEGER)
    RETURNS VARCHAR(1000)
    LANGUAGE SQL
    NOT DETERMINISTIC
    NO EXTERNAL ACTION
RETURN SUBSTR(sir_, 1, cate_);

-- Functia RIGHT_ extrage ultimele cate_ caractere dintr-un sir_
CREATE FUNCTION RIGHT_ (sir_ VARCHAR(1000), cate_ INTEGER)
    RETURNS VARCHAR(1000)
    LANGUAGE SQL
    NOT DETERMINISTIC
    NO EXTERNAL ACTION
RETURN SUBSTR(sir_, LENGTH(sir_) - cate_ + 1, cate_);
```

Cu totul altfel stau lucrurile dacă ne apucăm de redactarea funcţiei INITCAP_, care conţine variabile, structuri alternative şi repetitive. Figura 16.12 conţine corpul

funcţiei ce respectă întocmai sintaxa SQL PL, dar şi o mică parte din lungul mesaj de eroare afişat.

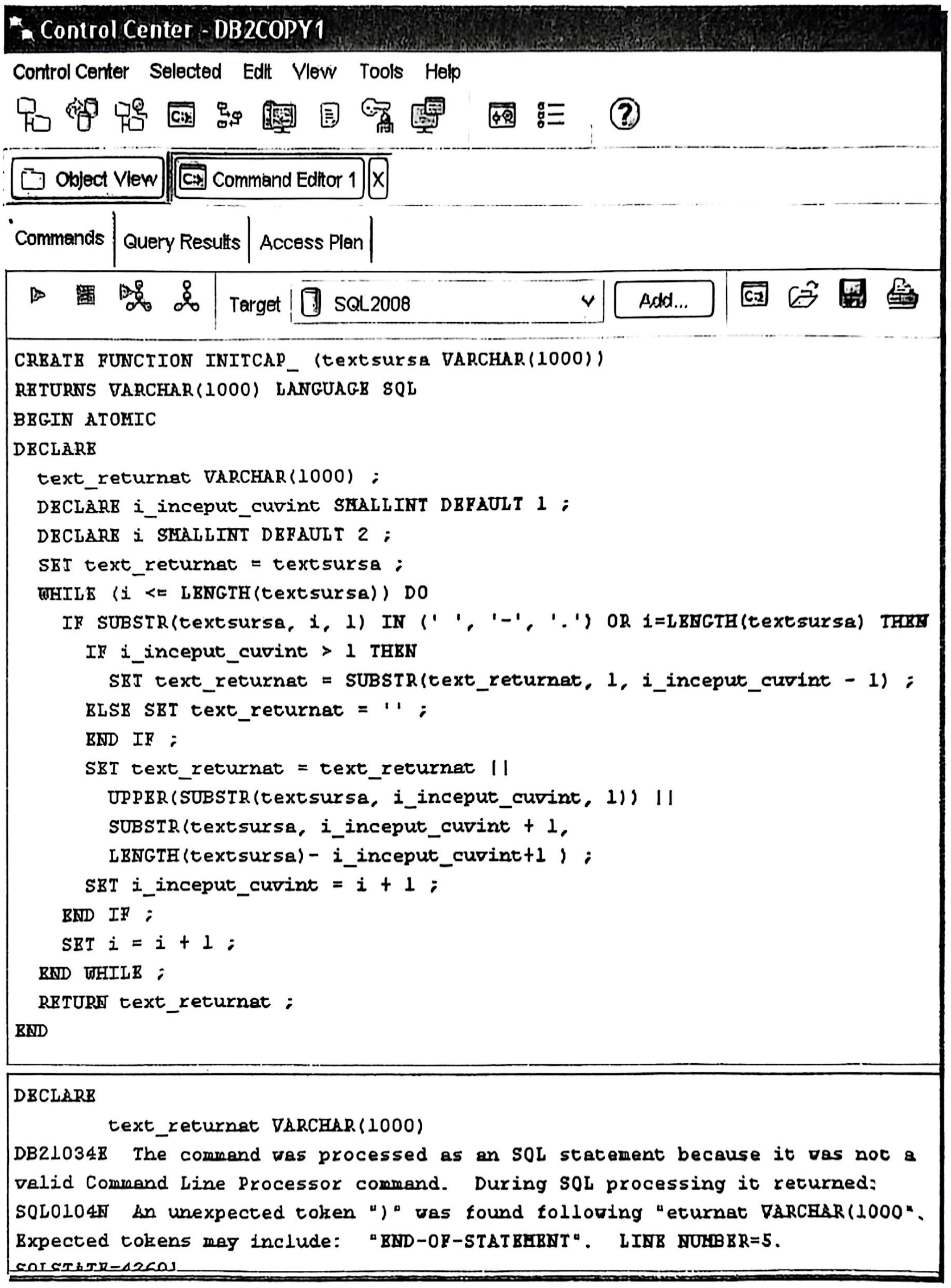


Figura 16.12. Refuzul DB2 Control Center de a recunoaşte funcţia în formatul „structurat"

Înainte de a vă apuca să căutaţi o eventuală eroare de sintaxă, trebuie să vă spun că aceeaşi funcţie, însă dispusă pe câteva rânduri, în care comenzile sunt dispuse pe aceeaşi linie una după alta, ca în listingul 16.18 funcţionează !

Listingul 16.18. Corpul funcţiei SQL PL ***INITCAP_***

```
CREATE FUNCTION INITCAP_ (textsursa VARCHAR(1000))
    RETURNS VARCHAR(1000) LANGUAGE SQL BEGIN ATOMIC
    DECLARE text_returnat VARCHAR(1000); DECLARE i_inceput_cuvant SMALLINT DEFAULT 1;
DECLARE i SMALLINT DEFAULT 2; SET text_returnat = textsursa; WHILE (i <= LENGTH(textsursa))
DO IF SUBSTR(textsursa, i, 1) IN (' ', '-', '.') OR i=LENGTH(textsursa) THEN IF i_inceput_cuvant > 1
```

```
THEN SET text_returnat = SUBSTR(text_returnat, 1, I_Inceput_cuvant - 1); ELSE SET text_returnat =
''; END IF; SET text_returnat = text_returnat || UPPER(SUBSTR(textsursa, I_Inceput_cuvant, 1)) ||
SUBSTR(textsursa, I_Inceput_cuvant + 1, LENGTH(textsursa) - I_Inceput_cuvant+1); SET
i_inceput_cuvant = I + 1; END IF; SET I = I + 1; END WHILE; RETURN text_returnat; END
```

În fapt, funcția este dispusă pe doar patru linii (vezi figura 16.13), însă listingul conține mai multe întrucât textul a fost rupt pentru a putea fi vizualizat.

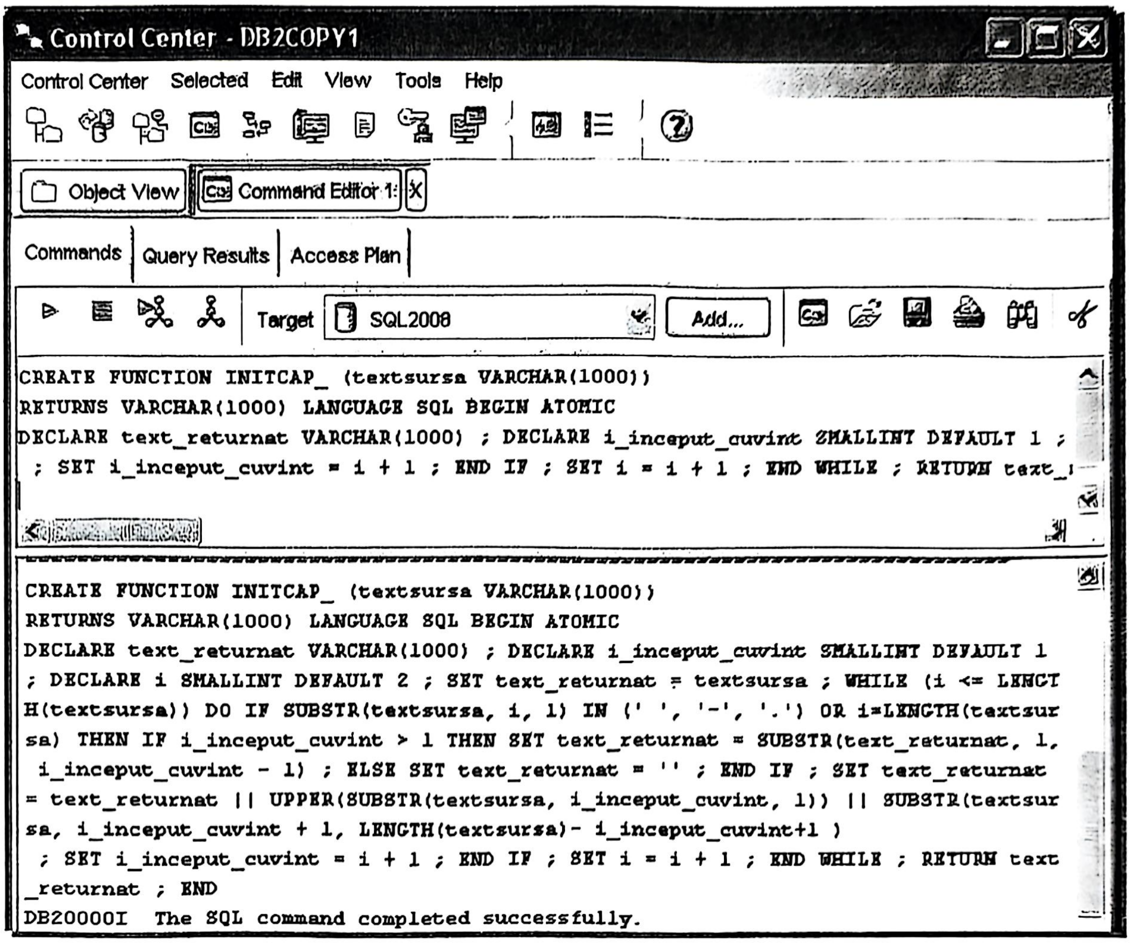


Figura 16.13. Forma funcției INITCAP_ acceptată de năzurosul DB2 Control Center

Din acest moment, sătui de fițele *DB2 Control Center*-ului, vom folosi exclusiv, în materie de funcții, proceduri și declanșatoare, *IBM Data Studio*. Cu riscul de a vă obosi, în listingul 16.19 includem varianta finală, corectă și gestionabilă de noua preferință IBM.

Listingul 16.19. Varianta finală a funcției ***INITCAP_*** *editată/compilată în IBM Data Studio*

```
CREATE FUNCTION INITCAP_(TEXTSURSA VARCHAR(1000))
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
DECLARE TEXT_RETURNAT VARCHAR(1000);
DECLARE I_INCEPUT_CUVANT SMALLINT DEFAULT 1;
DECLARE I SMALLINT DEFAULT 2;
SET TEXT_RETURNAT = LOWER(TEXTSURSA);
WHILE (I <= LENGTH(TEXTSURSA)) DO
    IF SUBSTR(TEXTSURSA, I, 1) IN (' ', '-', '.') OR I = LENGTH(TEXTSURSA) THEN
        IF I_INCEPUT_CUVANT > 1 THEN
```

```
            SET TEXT_RETURNAT = SUBSTR(TEXT_RETURNAT, 1, I_INCEPUT_CUVANT - 1);
        ELSE
            SET TEXT_RETURNAT = '';
        END IF;
        SET TEXT_RETURNAT = TEXT_RETURNAT ||
            UPPER(SUBSTR(TEXTSURSA, I_INCEPUT_CUVANT, 1)) ||
            LOWER(SUBSTR(TEXTSURSA, I_INCEPUT_CUVANT + 1,
                            LENGTH(TEXTSURSA) - I_INCEPUT_CUVANT + 1));
        SET I_INCEPUT_CUVANT = I + 1;
    END IF;
    SET I = I + 1;
END WHILE;
RETURN TEXT_RETURNAT;
END
```

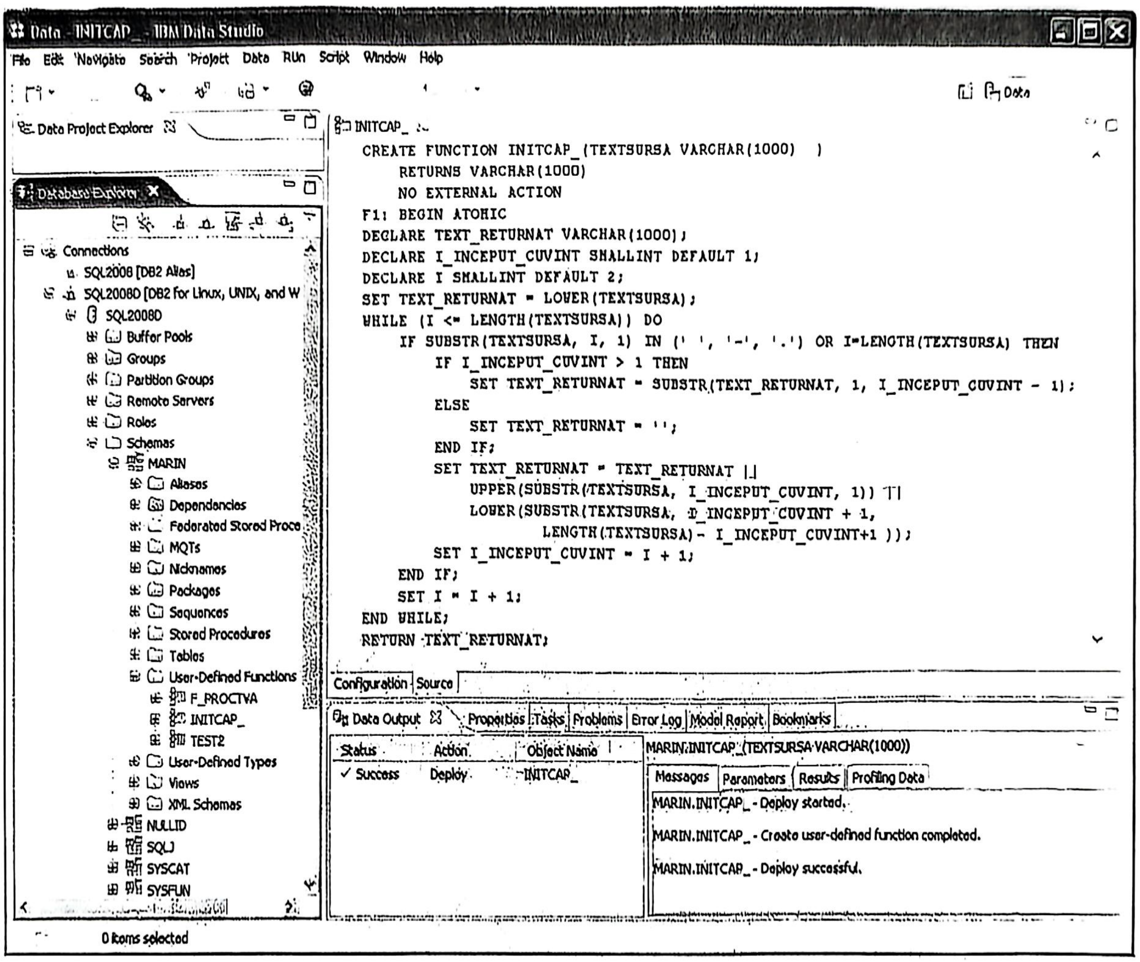


Figura 16.14. Crearea funcţiei INITCAP_ în IBM Data Studio

Faptul că sintaxa este operaţională este indicat în figura 16.14. Atenţie, lucrul în Data Studio este mai dificil la început (spre exemplu, pentru a compila şi a salva funcţia într-o subschemă se foloseşte opţiunea *Deploy*).

Listingul 16.20 conţine sintaxa funcţiei *f_data_scadenta* compilată şi plasată în schema curentă a bazei cu IBM Data Studio. Logica este aceeaşi ca în versiunile procedurale din celelalte trei servere, doar că unele comenzi (DECLARE, WHILE) şi funcţii (DAYOFWEEK_ISO) au un format uşor diferit.

Listingul 16.20. Funcția ***f_data_scadenta***

```
CREATE FUNCTION F_DATA_SCADENTA (DATAI_ DATE, NRZILE_SCADENTA_ SMALLINT)
    RETURNS DATE
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE ZI_CRT DATE;
    DECLARE V_GATA SMALLINT DEFAULT 0;
    DECLARE V_NR_ZILE_LUCRATOARE SMALLINT DEFAULT 0;
    SET ZI_CRT = DATAI_;
    WHILE (V_GATA = 0) DO SET ZI_CRT = ZI_CRT + 1 DAY;
        IF DAYOFWEEK_ISO (ZI_CRT) NOT IN (6,7) THEN
                SET V_NR_ZILE_LUCRATOARE = V_NR_ZILE_LUCRATOARE + 1;
                IF V_NR_ZILE_LUCRATOARE >= NRZILE_SCADENTA_ THEN
                        SET V_GATA = 1;
                END IF;
        END IF;
    END WHILE;
    RETURN ZI_CRT;
END
```

La invocarea funcției, dorind să aflăm data scadentă a fiecărei facturi în care numărul de zile este 19, DB2-ul mai are o pretenție: declararea explicită a numărului 19 ca SMALLINT printr-o funcție CAST:

SELECT NrFact, DataFact, marin.f_data_scadenta (facturi.DataFact,

CAST (19 AS SMALLINT)) AS "Scadenta (19 zile lucratoare)"

FROM facturi

Cele două funcții care asigură afișarea denumirii produselor prezente într-o factură sunt conținute în listingul 16.21. Remarcăm formatul de atribuire a rezultatului unei interogări unei variabile.

Listingul 16.21. Funcțiile SQL PL f_codpr_liniefact și f_produse_din_factura_v1

```
CREATE FUNCTION f_codpr_liniefact (nrfact_ INTEGER, linie_ SMALLINT)
    RETURNS SMALLINT
BEGIN ATOMIC
    DECLARE v_codpr SMALLINT DEFAULT 0;
    SET v_codpr = (SELECT CodPr FROM liniifact WHERE NrFact=nrfact_ AND Linie=linie_);
    RETURN v_codpr;
END
-- ===========================================================
CREATE FUNCTION f_produse_din_factura_v1 (nrfact_ INTEGER)
    RETURNS VARCHAR (1000)
BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '';
    DECLARE i SMALLINT DEFAULT 0;
    WHILE (i <= 99) DO
        SET i = i + 1;
        IF f_codpr_liniefact (nrfact_, i) = 0 THEN
                SET i=100;
        ELSE
                SET v_sir = v_sir || ' * ' || f_denpr( f_codpr_liniefact (nrfact_, i));
        END IF;
    END WHILE;
    RETURN v_sir;
END
```

16.2. Cursoare

SQL lucrează cu seturi de înregistrări. Noţiunea de *înregistare/linie curentă* nu există nici în modelul relaţional, nici în SQL. Cu toate acestea, aplicaţiile complexe necesită proceduri şi funcţii în care prelucrarea pe seturi nu este suficientă, oricât de puternice ar fi dialectele SQL în materie de subconsultări, opţiuni OLAP, ierarhii/recursivitate etc. Cursorul reprezintă un instrument ideal pentru prelucrările linie cu linie. De asemenea, în aplicaţiile client/server şi web, folosind funcţii bazate pe cursoare, fluxul datelor în reţea poate fi optimizat sensibil.

```
DECLARE
    -- cursorul se declară printr-o frază SELECT
    CURSOR c_cursor IS
        SELECT ...

    -- se declară variabilă compusă ce va stoca o linie a cursorului
    rec_cursor c_cursor%ROWTYPE
 ...
BEGIN
    ...
    OPEN c_cursor          /* la deschidere se executa fraza SELECT-definitie
                           şi se rezervă o zonă de memorie pentru liniile extrase */

    FETCH c_cursor INTO rec_cursor      /* se încarcă prima linie din cursor
                                        în variabila rec_cursor */

    WHILE c_cursor%FOUND LOOP           /* se execută bucla atâta timp cât se reuşeşte
                                        încărcarea unei alte linii din cursor */
        ...
        -- corpul buclei
        ...

        FETCH c_cursor INTO rec_cursor  /* se încearcă încarcarea următoarei linii din cursor;
                                        dacă nu s-a putut, rezultă că înregistrările cursorului
                                        sunt epuizate, iar c_cursor%FOUND are
                                        valoarea logică FALSE */
    END LOOP ;
    CLOSE c_cursor          /* de multe ori, un lucru deschis trebuie închis la loc */
    ....
END ;
```

Figura 16.15. Prima variantă de lucru cu un cursor

Un cursor reprezintă un set de înregistrări obţinut printr-o interogare şi stocat undeva în memorie, set ce poate fi parcurs secvenţial, uneori într-o singură direcţie (de la prima înregistrare la ultima), alteori în ambele direcţii, fiecare înregistrare păstrând, în unele cazuri, legătura cu linia tabelei din care provine.

Operaţiunile derulate în lucrul cu un cursor sunt:

- definirea/declararea;
- deschiderea;
- încarcărea uneia sau mai multor înregistrări din cursor;
- închiderea.

Chiar dacă este particularizată pe sintaxa Oracle PL/SQL, figura 16.15 ilustrează destul de bine prima (şi cea mai complicată) modalitate de folosire a

unui cursor. Definirea se face în zona declarativă. Variabila *rec_cursor* este folosită la stocarea înregistrării curente la trecerea prin cursor.

În zona executabilă a blocului mai întâi se deschide cursorul (OPEN). Deschiderea echivalează cu execuția interogării de la declararea cursorului și stocarea setului de linii obținut prin interogare într-o zonă de memorie. O comandă FETCH încarcă următoarea înregistrare din cursor în variabila *rec_cursor.* Structura repetitivă *WHILE c_cursor%FOUND LOOP ... END LOOP* asigură parcurgerea tuturor înregistrărilor cursorului, iar comanda CLOSE face curățenie, prin închidere, spațiul de memorie alocat fiind eliberat.

```
DECLARE
    -- cursorul se declară printr-o frază SELECT
    CURSOR c_cursor IS
        SELECT...

    -- nu mai este necesară declararea variabilei compozite REC_CURSOR
...
BEGIN
...
    FOR rec_cursor IN c_cursor  LOOP /* automat se execută și deschiderea
                                          și încarcarea înregistrării */
        ...
        ... -- corpul buclei
        ...

        -- citirea următoarei înregistrări se realizează automat
    END LOOP ;
    /* de multe ori, un lucru deschis trebuie închis la loc. NU ȘI DE DATA ACEASTA ! */
    ...
END ;
```

Figura 16.16. A doua variantă de lucru cu un cursor

A doua variantă de lucru, a cărei logică este cea din figura 16.16 este mai simplă întrucât:

- cursorul se declară ad-hoc în zona executabilă;
- variabila care păstrează înregistrarea curentă din cursor nu trebuie definită în zona DECLARE;
- cursorul nu mai trebuie deschis și închis explicit;
- deplasarea la înregistrarea următoare este automată.

Există și o treia variantă, derivată din a doua și mai simplă decât aceasta, în care declararea cursorului se face direct în structura iterativă de parcurgere a înregistrărilor (vezi figura 16.17).

```
DECLARE
    -- nu mai este necesară declararea variabilei compozite REC_CURSOR
..
BEGIN
...
    FOR rec_cursor IN (SELECT ... ) LOOP /* cursorul se declară ad-hoc si, simultan,
                                         /* se deschide şi se încarcarcă prima înregistrare */

            -- corpul buclei


          -- citirea următoarei înregistrări se realizează automat
    END LOOP ;
    ...
END ;
```

Figura 16.17. A treia variantă de lucru cu un cursor

Deşi sunt câteva diferenţe de sintaxă, pe care le vom discuta pe scurt în continuare, logica celor trei variante este implementată în toate cele patru limbaje ale serverelor BD din această lucrare. De asemenea, în paragrafele dedicate procedurilor (16.5.1 - 16.5.4) vom completa discuţia referitoare la cursoarele explicite.

16.2.1. Cursoare în PostgreSQL PL/pgSQL

Începem exemplificarea cursoarelor printr-o funcţie ce rezolvă o problemă din paragraful 16.1.1 - cea care îşi propunea să obţină raportul din figura 16.5. În locul celor două funcţii din listingul 16.7, putem folosi una singură - *f_produse_din_factura_v2* - pe care o prezentăm în listingul 16.22. Ca element de noutate apare şi o variabilă de tip RECORD, *rec_linii*. Această este grozav de versatilă, deoarece poate stoca o linie din orice tabelă, neavând o structură predefinită. În cazul nostru, *rec_linii* va stoca o linie a cursorului ad-hoc, definit prin SELECT-ul din structura *FOR*... Practic, cursorul va conţine atâtea înregistrări câte linii sunt în tabela LINIIFACT pentru factura cu numărul *nrfact_*.

Listingul 16.22. Funcţia ***f_produse_din_factura_v2***

```
CREATE OR REPLACE FUNCTION f_produse_din_factura_v2 (nrfact_ facturi.nrfact%TYPE)
    RETURNS TEXT
AS $$
DECLARE
    rec_linii RECORD;
    v_sir TEXT := '';
BEGIN
    FOR rec_linii IN (SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ ) LOOP
        v_sir := v_sir || ' * ' || f_denpr(rec_linii.CodPr);
    END LOOP;
    RETURN v_sir;
END;
$$ LANGUAGE plpgsql;
```

Fireşte că apelul funcţiei nu pune absolut nicio problemă:

```
SELECT facturi.NrFact, facturi.DataFact,
        f_produse_din_factura_v2(facturi.NrFact) AS Lista_produse
```

FROM facturi

ORDER BY 1

Ne propunem în continuare să creăm funcțiile stocate care să returneze expresiile cheilor primare și străine, indiferent de numărul atributelor din cheile compuse (în paragraful 15.1.1 am presupus că pot fi maximum patru atribute într-o cheie compusă). Funcția *f_cheie_primară* din listingul 16.23 extrage mai întâi numele restricției de tip cheie primară pentru tabela dată. Știm că o tabelă nu poate avea mai multe chei primare, așa că nu avem nicio grijă la interogarea tabelei virtuale dicționar *information_schema.table_constraints*. Cursorul va prelua toate înregistrările extrase din view-ul *information_schema.key_column_usage* pentru restricția cu numele stocat în variabila *v_nume_restricție*.

Listingul 16.23. Funcția ***f_cheie_primara***

```
CREATE OR REPLACE FUNCTION f_cheie_primara (tabela_ TEXT)
    RETURNS TEXT AS $$
DECLARE
    v_nume_restrictie TEXT;
    v_sir VARCHAR(200) := '(';
    rec_atribute RECORD;
BEGIN
    SELECT constraint_name INTO v_nume_restrictie
    FROM information_schema.table_constraints
    WHERE UPPER(table_name)=UPPER(tabela_) AND constraint_type='PRIMARY KEY';

    FOR rec_atribute IN (       SELECT column_name, ordinal_position
                                FROM information_schema.key_column_usage
                                WHERE constraint_name = v_nume_restrictie
                                ORDER BY ordinal_position) LOOP
        v_sir := v_sir || CASE WHEN LENGTH(v_sir)=1 THEN '' ELSE ',' END
                || rec_atribute.column_name;
    END LOOP;
    RETURN v_sir || ')';
END;
$$ LANGUAGE plpgsql;
```

La fiecare înregistrare a cursorului, variabila *v_șir* va fi „incrementată" cu atributul corespunzător liniei curente a cursorului. Testăm funcția pentru tabela PERSCLIENȚI (rezultatul fiind cel din figura 16.18):

SELECT f_cheie_primara ('persclienti')

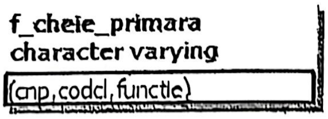


Figura 16.18. Componența cheii primare a tabelei PERSCLIENȚI

Căutând un pretext pentru folosirea în tandem a două cursoare, ne propunem să construim o funcție care să afișeze toate expresiile cheilor alternative pentru o tabelă dată. Diferența principală față de problema anterioară este că o tabelă poate avea mai multe chei alternative (și doar o singură cheie primară). Este și motivul pentru care funcția (vezi listingul 16.24) se numește *f_chei_alternative*.

Listingul 16.24. Funcţie PL/pgSQL ce foloseşte două curşoare

```
CREATE OR REPLACE FUNCTION f_chei_alternative (tabela_ TEXT)
    RETURNS TEXT AS
$$
DECLARE
    v_sir TEXT := '';
    rec_chei_alternative RECORD;
    rec_atribute RECORD;
BEGIN
    -- este posibil ca intr-o tabela sa avem mai multe chei alternative, asa ca
    -- folosim un prim cursor
    FOR rec_chei_alternative IN (SELECT constraint_name
                        FROM information_schema.table_constraints
                        WHERE UPPER(table_name) = UPPER(tabela_)
                                AND constraint_type = 'UNIQUE') LOOP
        v_sir := v_sir || ' UNIQUE (';
        -- fiecare cheie alternativa poate fi compusa, asa ca folosim
        -- un al doilea cursor
        FOR rec_atribute IN (     SELECT column_name, ordinal_position
                                FROM information_schema.key_column_usage
                                WHERE constraint_name =
                                        rec_chei_alternative.constraint_name
                                ORDER BY ordinal_position) LOOP
            v_sir := v_sir || CASE WHEN
                        SUBSTR(v_sir, LENGTH(v_sir)-7,9) = 'UNIQUE ('THEN '' ELSE ',' END
                        || rec_atribute.column_name;
        END LOOP;
    END LOOP;
    RETURN v_sir || CASE v_sir WHEN '' THEN '' ELSE ')' END;
END;
$$ LANGUAGE plpgsql;
```

Ar trebui ca explicaţiile inserate în corpul funcţiei să uşureze înţelegerea mecanismului de execuţie. Partea cea mai interesantă este „manevrarea" variabilei *v_şir*. Structura CASE din corpul buclei verifică dacă atributul curent (*rec_atribute.column_name*) este primul din componenţa cheii alternative curente (*rec_chei_alternative.constraint_name*). Dacă nu e primul, trebuie precedat de o virgulă.

La capitolul specificităţi „cursoristice" PL/pgSQL, ar trebui adăugat că sintaxa comenzii FETCH este foarte generoasă. Astfel, putem încărcă direct ultima înregistrare din cursor (prin FETCH LAST) sau reîncărca precedenta înregistrare (prin FETCH RELATIVE - 1). Mai mult, există comanda MOVE prin care ne putem deplasa în cursor după bunul plac, fără a încărca noua înregistrare curentă.

16.2.2. Cursoare în Oracle PL/SQL

În Oracle, cursoarele se împart în implicite şi explicite. Cele implicite furnizează câteva informaţii despre execuţia comenzilor DML (de exemplu, câte linii au fost şterse în urma execuţiei unei comenzi DELETE) şi nu ne interesează în această lucrare. Cursoarele explicite funcţionează după logica expusă în deschiderea paragrafului 16.2.

Începem cu funcţia corespondentă primeia dintre cele descrise în paragraful anterior (listingul 16.25). Tipul returnat de funcţie este VARCHAR2, iar variabila ce stochează înregistrarea curentă din cursor nu trebuie declarată explicit.

Listingul 16.25. Funcţia PL/SQL ***f_produse_din_factura_v2***

```
CREATE OR REPLACE FUNCTION f_produse_din_factura_v2 (nrfact_ facturi.nrfact%TYPE)
    RETURN VARCHAR2
AS
    v_sir VARCHAR2(1000) := '';
BEGIN
    FOR rec_linii IN (SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ ) LOOP
        v_sir := v_sir || ' • ' || f_denpr(rec_linii.CodPr);
    END LOOP;
    RETURN v_sir;
END;
```

Singurele elemente importante ce diferenţiază funcţia *f_cheie_primară* (vezi listingul 16.25) de omoloaga sa din PL/pgSQL (listingul 16.23) este denumirea tabelelor virtuale din dicţionarul de date şi atributelor acestora.

Listingul 16.25. Funcţia PL/SQL ***f_cheie_primara***

```
CREATE OR REPLACE FUNCTION f_cheie_primara (tabela_ VARCHAR2)
    RETURN VARCHAR2
AS
    v_nume_restrictie VARCHAR2(40);
    v_sir VARCHAR(1000) := '(';
BEGIN
    SELECT constraint_name INTO v_nume_restrictie
    FROM user_constraints
    WHERE UPPER(table_name) = UPPER(tabela_) AND constraint_type = 'P';

    FOR rec_atribute IN (           SELECT column_name, position
                                    FROM user_cons_columns
                                    WHERE constraint_name = v_nume_restrictie
                                    ORDER BY position) LOOP
        v_sir := v_sir || CASE WHEN LENGTH(v_sir) = 1 THEN '' ELSE ',' END
                || rec_atribute.column_name;
    END LOOP;
    RETURN v_sir || ')';
END;
```

Analog stau lucrurile şi pentru funcţia f_chei_alternative din listingul 16.27.

Listingul 16.27. Funcţia PL/SQL ***f_chei_alternative***

```
CREATE OR REPLACE FUNCTION f_chei_alternative (tabela_ VARCHAR2)
    RETURN VARCHAR2
AS
    v_sir VARCHAR2(1000) := ' ';
BEGIN
    -- este posibil ca intr-o tabela sa avem mai multe chei alternative, asa ca
    -- folosim un prim cursor
    FOR rec_chei_alternative IN (SELECT constraint_name
                        FROM user_constraints
                        WHERE UPPER(table_name) = UPPER(tabela_)
                         AND constraint_type = 'U') LOOP
        v_sir := v_sir || ' UNIQUE (';
```

```
        -- fiecare cheie alternativa poate fi compusa, asa ca folosim
        -- un al doilea cursor
        FOR rec_atribute IN (       SELECT column_name, position
                                    FROM user_cons_columns
                                    WHERE constraint_name =
                                          rec_chei_alternative.constraint_name
                                    ORDER BY position) LOOP
            v_sir := v_sir || CASE WHEN
                          SUBSTR(v_sir, LENGTH(v_sir)-7,9) = 'UNIQUE (' THEN '' ELSE ',' END
                          || rec_atribute.column_name;
        END LOOP;
    END LOOP;
    RETURN v_sir || CASE v_sir WHEN ' ' THEN ''  ELSE ')' END;
END;
```

16.2.3. Cursoare în SQL Server Transact-SQL

În Transact-SQL, cursorul nu poate fi definit ad-hoc, ci printr-o comandă DECLARE (vezi listingul 16.28). Schema de folosire este aceeaşi, singura diferenţă fiind funcţia @@FETCH_STATUS, care furnizează informaţii despre cum a decurs ultima comandă FETCH executată. Astfel, dacă valoarea returnată este 0, atunci încărcarea s-a făcut cu succes, aceasta fiind, de fapt, condiţia de repetare a buclei. În Transact-SQL, cursorul poate fi parcurs în ambele direcţii, dar deocamdată nu avem niciun motiv să apelăm la serviciile acestei opţiuni.

Listingul 16.28. Funcţia T-SQL ***f_produse_din_factura_v2***

```
CREATE FUNCTION f_produse_din_factura_v2 (@nrfact INT)
    RETURNS VARCHAR(1000) AS
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_codpr SMALLINT
    SET @v_sir = ''
    DECLARE c_linii CURSOR FOR SELECT CodPr FROM liniifact WHERE NrFact = @nrfact
    OPEN c_linii
    FETCH NEXT FROM c_linii INTO @v_codpr
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + ' * ' + dbo.f_denpr(@v_codpr)
        FETCH NEXT FROM c_linii INTO @v_codpr
    END
    CLOSE c_linii
    DEALLOCATE c_linii
    RETURN @v_sir
END
```

Listingul 16.29 conţine echivalentul T-SQL al funcţiilor PL/pgSQL şi PL/SQL prezentate în paragrafele anterioare, prin care se furnizează expresia cheii primare a unei tabele-argument.

Listingul 16.29. Funcţia T-SQL ***f_cheie_primară***

```
IF OBJECT_ID (N'dbo.f_cheie_primara', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_cheie_primara;
GO
CREATE FUNCTION f_cheie_primara (@tabela VARCHAR(40))
    RETURNS VARCHAR(1000) AS
```

```
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_nume_restrictie VARCHAR(40)
    SET @v_sir = '(';

    SELECT @v_nume_restrictie = constraint_name
    FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
    WHERE UPPER(table_name) = UPPER(@tabela) AND constraint_type = 'PRIMARY KEY';

    DECLARE @v_atribut VARCHAR(40)
    DECLARE @v_pozitie TINYINT

    DECLARE c_linii CURSOR FOR SELECT column_name, ordinal_position
                               FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
                               WHERE constraint_name = @v_nume_restrictie ORDER BY 2
    OPEN c_linii
    FETCH NEXT FROM c_linii INTO @v_atribut, @v_pozitie
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + CASE WHEN LEN(@v_sir) = 1 THEN '' ELSE ',' END +
                @v_atribut;
        FETCH NEXT FROM c_linii INTO @v_atribut, @v_pozitie
    END
    CLOSE c_linii
    DEALLOCATE c_linii
    RETURN @v_sir + ')'
END
```

Funcția următoare – *f_chei_alternative* – prezentată în listingul 16.30 ilustrează modul în care putem folosi simultan două cursoare, *c_chei_alternative* și *c_atribute*. Față de variantele PL/(pg)SQL, în T-SQL, cel de-al doilea cursor este declarat, deschis, parcurs și închis pentru fiecare înregistrare din primul cursor.

Listingul 16.30. Funcția T-SQL ***f_chei_alternative***

```
IF OBJECT_ID (N'dbo.f_chei_alternative', N'FN') IS NOT NULL
  DROP FUNCTION dbo.f_chei_alternative;
GO
CREATE FUNCTION f_chei_alternative (@tabela VARCHAR(40))
    RETURNS VARCHAR(1000) AS
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_nume_restrictie VARCHAR(100)
    DECLARE @v_atribut VARCHAR(100)
    DECLARE @v_pozitie TINYINT
    SET @v_sir = ' ';

    -- primul cursor
    DECLARE c_chei_alternative CURSOR FOR
        SELECT constraint_name
        FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
        WHERE UPPER(table_name) = UPPER(@tabela) AND constraint_type = 'UNIQUE'

    OPEN c_chei_alternative
    FETCH NEXT FROM c_chei_alternative INTO @v_nume_restrictie
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + ' UNIQUE ('

        -- al doilea cursor
        DECLARE c_atribute CURSOR FOR
```

```
                        SELECT column_name, ordinal_position
                        FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
                        WHERE constraint_name = @v_nume_restrictie
                        ORDER BY 2
        OPEN c_atribute
        FETCH NEXT FROM c_atribute INTO @v_atribut, @v_pozitie
        WHILE @@FETCH_STATUS = 0 BEGIN
                SET @v_sir = @v_sir + CASE WHEN
                        SUBSTRING(@v_sir, LEN(@v_sir)-7,9) = 'UNIQUE ('
                                THEN '' ELSE ',' END + @v_atribut
                FETCH NEXT FROM c_atribute INTO @v_atribut, @v_pozitie
        END
        CLOSE c_atribute
        FETCH NEXT FROM c_chei_alternative INTO @v_nume_restrictie
    END

    CLOSE c_chei_alternative
    DEALLOCATE c_atribute
    DEALLOCATE c_chei_alternative

    RETURN @v_sir + CASE @v_sir WHEN ' ' THEN ' ' ELSE ')' END

END
```

16.2.4. Cursoare în DB2 SQL PL

Cea mai simplă variantă a funcţiei care primeşte un număr de factură şi returnează un şir de caractere ce conţine denumirile tuturor produselor din factura respectivă este cea din listingul 16.31. Definirea cursorului s-a realizat prin secvenţa *FOR rec1... END FOR.*

Listingul 16.31. Funcţia SQL PL ***f_produse_din_factura_v2***

```
CREATE FUNCTION F_PRODUSE_DIN_FACTURA_V2 (nrfact_ INTEGER)
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '';
    FOR rec1 AS SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ DO
        SET v_sir = v_sir || ' * ' ||
                CAST (F_DENPR(CAST (rec1.CodPr AS SMALLINT)) AS VARCHAR(30));
    END FOR;
    RETURN v_sir;
END
```

Continuăm, în tradiţia paragrafelor precedente, cu funcţia care furnizează expresia cheii primare pentru o tabelă dată (parametru) (vezi listingul 16.32). Prima comandă de atribuire (SET) foloseşte, pe post de expresie, o interogare. Rezultatul interogării va fi atribuit variabilei *v_nume_restricţie*.

Listingul 16.32. Funcţia SQL PL ***f_cheie_primară***

```
CREATE FUNCTION f_cheie_primara(tabela_ VARCHAR(40))
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '(';
```

```
    DECLARE v_nume_restrictie VARCHAR(40) ;
    DECLARE v_atribut VARCHAR(40);
    DECLARE v_pozitie SMALLINT;

    SET v_nume_restrictie = (SELECT constraint_name INTO
                    FROM sysibm.table_constraints
                    WHERE table_schema = CURRENT_USER AND
                         UPPER(table_name) = UPPER(tabela_)
                         AND constraint_type = ' PRIMARY KEY' );
    FOR rec_linii AS SELECT colname, colseq
                         FROM SYSCAT.keycoluse
                         WHERE constname = v_nume_restrictie ORDER BY 2 DO
        SET v_sir = v_sir || CASE WHEN LENGTH(v_sir) = 1 THEN '' ELSE ',' END
             || rec_linii.colname;
    END FOR;
    RETURN v_sir || ')'    ;
END
```

Nici funcția *f_chei_alternative* (listingul 16.33), prin care exemplificăm includerea unei structuri iterative definită pe baza unui cursor în altă structură iterativă corespunzătoare altui cursor, nu aduce noutăți remarcabile.

Listingul 16.33. Funcția SQL PL ***f_chei_alternative***

```
CREATE FUNCTION f_chei_alternative(tabela_ VARCHAR(40))
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '';
    DECLARE v_nume_restrictie VARCHAR(40) ;
    DECLARE v_atribut VARCHAR(40);
    DECLARE v_pozitie SMALLINT;

    FOR rec_chei_altern AS SELECT constraint_name FROM sysibm.table_constraints
              WHERE table_schema = CURRENT_USER AND
                    UPPER(table_name) = UPPER(tabela_)
                    AND constraint_type = ' UNIQUE' DO
        SET v_sir = v_sir || ' UNIQUE (' ;
        FOR rec_linii AS SELECT colname, colseq
                    FROM SYSCAT.keycoluse
                    WHERE constname = rec_chei_altern.constraint_name ORDER BY 2
                         DO
              SET v_sir = v_sir || CASE WHEN
                    SUBSTR(v_sir, LENGTH(v_sir) -7,9) = 'UNIQUE ('
                         THEN '' ELSE ',' END || rec_linii.colname;
        END FOR;
    END FOR;
    RETURN v_sir || CASE v_sir WHEN '' THEN '' ELSE ')' END;
END
```

16.3. Funcții ce furnizează seturi de înregistrări

Un „debușeu" clasic al funcțiilor stocate este obținerea de rapoarte dintre cele mai diverse. Specific acestor funcții este că returnează seturi de înregistrări dintr-o tabelă/interogare, set furnizat interfeței sau logicii aplicației de către serverul BD.

Vom vedea că sunt diferenţe semnificative, ba chiar, mai mult, presupuşii outsideri – PostgreSQL şi SQL Server - sunt, pe alocuri mai generoşi decât greii DB2 şi Oracle.

16.3.1. Funcţii ce returnează seturi de înregistrări în PostgreSQL

Începem cu o funcţie simplă - *f_facturi_filtrate_cronologic* - care operează o selecţie a liniilor tabelei FACTURI pentru un interval calendaristic (vezi listingul 16.34). Eleganţa funcţiei provine din clauza *RETURNS SET OF...*, care poate indica structura (atributele) unei tabele, tabele virtuale sau a unui tip-utilizator[4]. Funcţia nu este redactată în PL/pgSQL, ci în SQL, având doi parametri de tip DATE care sunt delimitatorii intervalului.

Listingul 16.34. Funcţia SQL (din PostgreSQL) ***f_facturi_filtrate_cronologic***

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate_cronologic
        (data_initiala DATE, data_finala DATE)
        RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi WHERE datafact BETWEEN $1 AND $2
$$ LANGUAGE SQL;
```

Odată compilată, funcţia poate fi invocată returnând un rezultat ca orice tabelă sau tabelă virtuală:

SELECT *

FROM f_facturi_filtrate_cronologic (DATE'2007-09-14', CURRENT_DATE) t

Similar, putem crea o funcţie care returneaza numai facturile unui anumit client (cu un cod specificat - vezi listingul 16.35).

Listingul 16.35. Funcţia SQL (din PostgreSQL) ***f_facturile_unui_client***

```
CREATE OR REPLACE FUNCTION f_facturile_unui_client
    (codcl_ clienti.codcl%TYPE) RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi WHERE codcl = $1
$$ LANGUAGE SQL;
```

Cele două funcţii pot fi folosite în aceeaşi interogare pentru aplicarea a două filtre. Spre exemplu, dorim extragerea tuturor facturilor emise clientului 1001 după 5 august 2007:

SELECT * FROM f_facturile_unui_client (1001) t

INTERSECT

SELECT *

FROM f_facturi_filtrate_cronologic (DATE'2007-08-05', CURRENT_DATE) t

Funcţia următoare - *f_facturi_filtrate* –, redactată tot în SQL, prezintă trei parametri: un cod de client, o dată iniţială şi o dată finală care delimitează un interval calendaristic (vezi listingul 16.36). Setul de înregistrări returnate reprezintă facturile destinate clientului dat (parametru) întocmite în intervalul definit de cele două date calendaristice. Dacă la invocarea funcţiei, vreunul dintre parametri este

4. Crearea şi utilizarea tipurilor-utilizator constituie subiectul capitolului 19.

NULL se consideră că nu se mai face filtrarea liniilor după parametrul respectiv. Astfel, când codul clientului este NULL, atunci se vor extrage facturile pentru toți clienții. Dacă data inițială este NULL, atunci se vor extrage facturile de la începuturile bazei de date (să zicem, 1 ianuarie 2005).

Listingul 16.36. Funcția SQL (din PostgreSQL) *f_facturi_filtrate*

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate (
codcl_ clienti.codcl%TYPE, data_initiala DATE, data_finala DATE)
    RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi
    WHERE codcl = COALESCE($1, codcl)
             AND datafact BETWEEN COALESCE($2, DATE'2005-01-01')
             AND COALESCE($3, DATE'2010-12-31')
$$ LANGUAGE SQL;
```

Iată și câteva modalități de folosire a funcției pentru a afla informații precum:

- facturile clientului 1001 emise între 14 septembrie 2007 și data curentă:

SELECT *
FROM f_facturi_filtrate(1001, DATE'2007-09-14', CURRENT_DATE) t

- toate facturile clientului 1001:

SELECT *
FROM f_facturi_filtrate(1001, NULL, NULL) t

- facturile emise pe 7 august 2007:

SELECT *
FROM f_facturi_filtrate(NULL, DATE'2007-08-07', DATE'2007-08-07') t

- toate facturile emise:

SELECT *
FROM f_facturi_filtrate(NULL, NULL, NULL) t

După cum sugeram ceva mai sus, în cazul în care setul de înregistrări returnat nu are exact structura unei tabele, trebuie creat un tip *tip_nou* și folosită clauza RETURNS SET OF *tip_nou*, după cum vom vedea în capitolul 19.

16.3.2. Funcții ce returnează seturi de înregistrări în Oracle PL/SQL

Oracle PL/SQL este cel mai zgârcit dintre cele patru limbaje în materie de obținere a unui set de înregistrări dintr-o funcție-utilizator. Există două soluții principale, plus una ocolitoare. Singura pe care o discutăm acum este cea bazată pe variabile cursor. Cea ocolitoare va fi discutată foarte pe scurt și în paragrafele 16.6 și 18.2. Soluția elegantă, care returnează valori-masiv (NESTED TABLE), va fi discutată abia în capitolul 19.

Funcția *f_facturi_filtrate1* din listingul 16.37 returnează o variabilă cursor – *set_of_facturi*. Clauza RETURN indică un tip „cât se poate" de general – SYS_REFCURSOR. Practic, funcția ar putea returna orice set de înregistrări. În zona executabilă variabila cursor este simultan declarată și populată prin comanda *OPEN... FOR...*

Listingul 16.37. Funcţia PL/SQL ***f_facturi_filtrate1***

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate1 (data_initiala DATE, data_finala DATE)
        RETURN SYS_REFCURSOR
IS
    set_of_facturi SYS_REFCURSOR;
BEGIN
    OPEN set_of_facturi FOR
        SELECT * FROM facturi WHERE datafact BETWEEN data_initiala AND data_finala;
    RETURN set_of_facturi;
END f_facturi_filtrate1;
```

Un pic mai dificil este modul de afişare în SQL Developer a rezultatului funcţiei la invocarea acesteia într-o consultare (vezi figura 16.19).

SELECT f_facturi_filtrate2(DATE'2007-08-15', CURRENT_DATE)

FROM dual

După cum sugerează figura, rezultatul iniţial este dispus pe o singură linie, fiind necesară selectarea liniei respective, activarea meniului contextual, alegerea opţiunii *Single Record View* şi apoi a butonului ... .

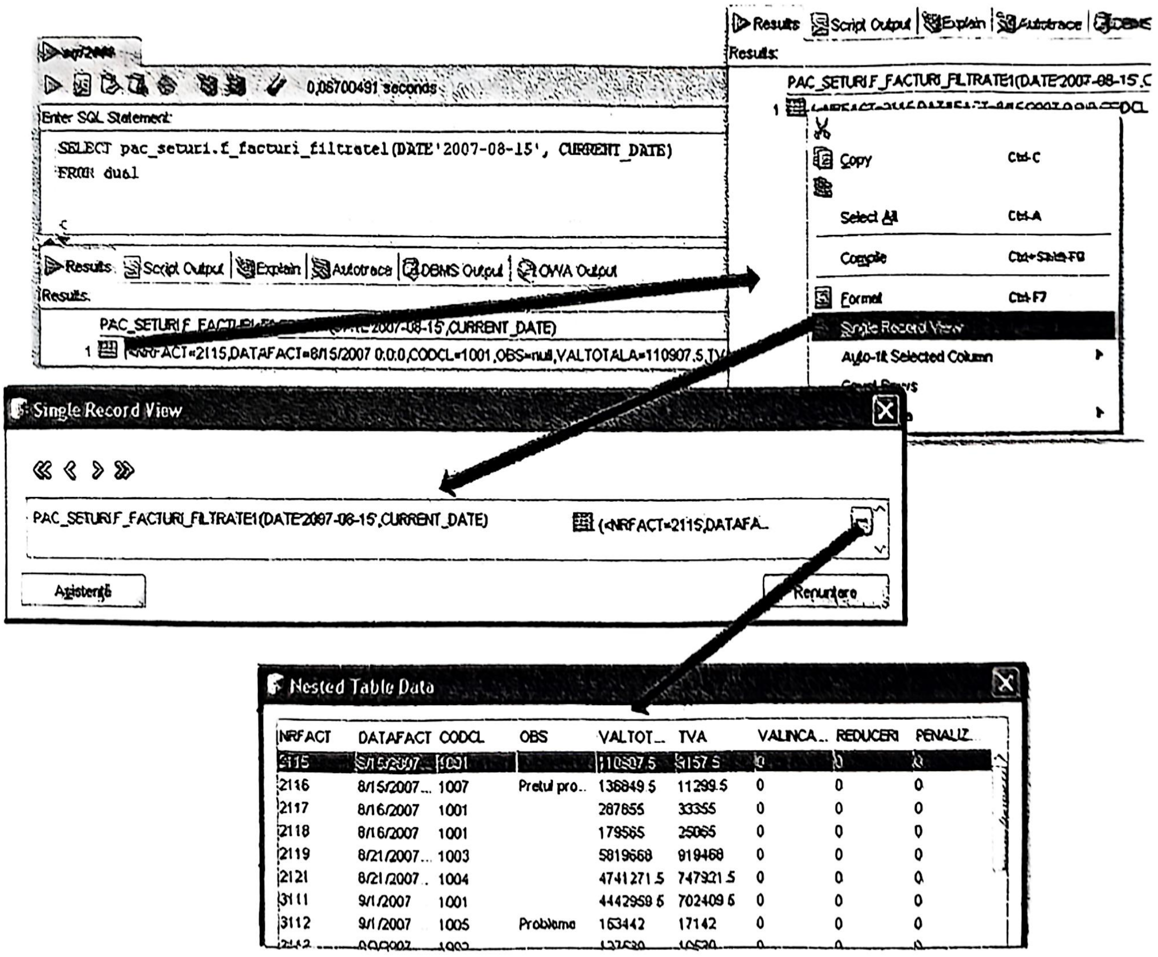
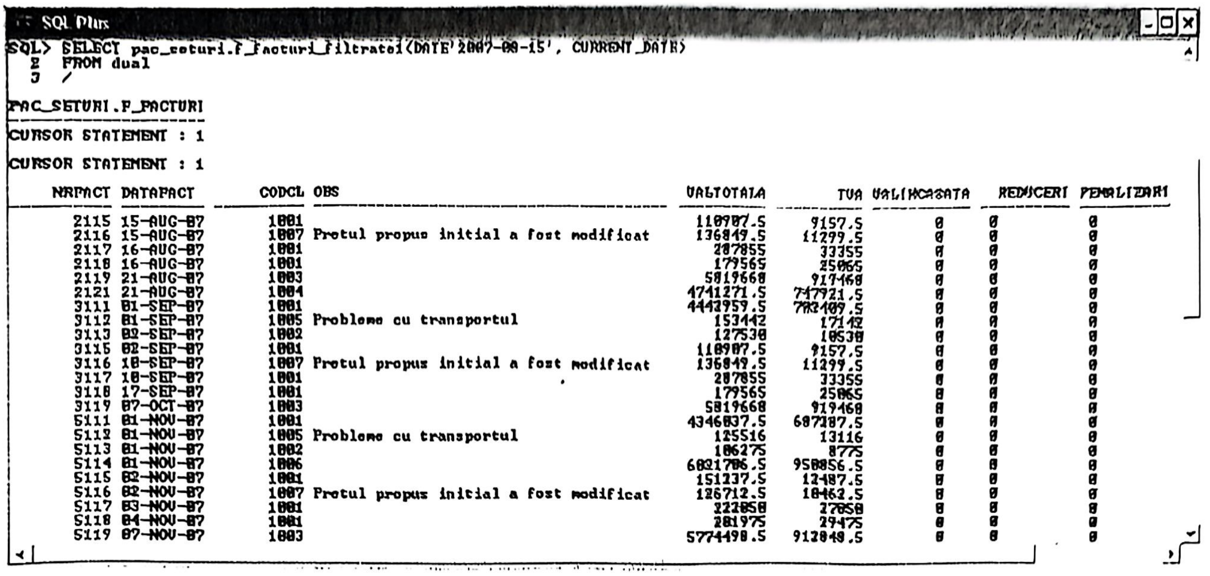


Figura 16.19. Vizualizarea în SQL Developer a setului de înregistrări furnizat de o funcţie ce foloseşte o variabilă cursor

Curios este că, deşi ciudată în SQL Developer, afişarea este corectă „din prima" în clientul clasic şi nesofisticat al Oracle - SQL*Plus (vezi figura 16.20). Trebuie spus, însă, că până la afişarea rezultatului ca în figură, în SQL*Plus trebuie

configurați o serie întreagă de parametri (mărimea ferestrei, lungimea unei linii, numărul de linii dintr-o pagină etc.), așa că nu vă lăsați înselați de aparențe.



```
SQL> SELECT pac_seturi.f_facturi_filtrate(DATE'2007-09-15', CURRENT_DATE)
  2  FROM dual
  3  /

PAC_SETURI.F_FACTURI
--------------------
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

   NRFACT DATAFACT       CODCL OBS                                        VALTOTALA        TVA VALINCASATA   REDUCERI PENALIZARI
--------- --------- ---------- ---------------------------------------- ---------- ---------- ----------- ---------- ----------
     2115 15-AUG-07       1001                                            110907.5     9157.5           0          0          0
     2116 15-AUG-07       1007 Pretul propus initial a fost modificat     136849.5    11299.5           0          0          0
     2117 16-AUG-07       1001                                              287855      33355           0          0          0
     2118 16-AUG-07       1001                                              179565      25065           0          0          0
     2119 21-AUG-07       1003                                             5819668     917468           0          0          0
     2121 21-AUG-07       1004                                           4741271.5   747921.5           0          0          0
     3111 01-SEP-07       1001                                           4443959.5   703409.5           0          0          0
     3112 01-SEP-07       1005 Probleme cu transportul                      153442      17142           0          0          0
     3113 02-SEP-07       1002                                              127530      10530           0          0          0
     3115 02-SEP-07       1001                                            110907.5     9157.5           0          0          0
     3116 10-SEP-07       1007 Pretul propus initial a fost modificat     136849.5    11299.5           0          0          0
     3117 10-SEP-07       1001                                              287855      33355           0          0          0
     3118 17-SEP-07       1001                                              179565      25065           0          0          0
     3119 07-OCT-07       1003                                             5819668     919468           0          0          0
     5111 01-NOV-07       1001                                           4346037.5   687287.5           0          0          0
     5112 01-NOV-07       1005 Probleme cu transportul                      125516      13116           0          0          0
     5113 01-NOV-07       1002                                              106275       8775           0          0          0
     5114 01-NOV-07       1006                                           6021706.5   950856.5           0          0          0
     5115 02-NOV-07       1001                                            151237.5    12487.5           0          0          0
     5116 02-NOV-07       1007 Pretul propus initial a fost modificat     126712.5    10462.5           0          0          0
     5117 03-NOV-07       1001                                              222858      27858           0          0          0
     5118 04-NOV-07       1001                                              281975      29475           0          0          0
     5119 07-NOV-07       1003                                           5774498.5   912849.5           0          0          0
```

Figura 16.20. Vizualizarea în SQL Plus a setului de înregistrări furnizat de o funcție ce folosește o variabilă cursor

16.3.3. Funcții ce returnează seturi de înregistrări în MS SQL Server

Transact-SQL este, probabil, cel mai generos dintre cele patru limbaje discutate în cartea de față în materie de opțiuni pentru funcțiile ce returnează seturi de înregistrări. Clauza *RETURNS TABLE* este cea care simplifică radical lucrurile. Să începem cu funcția *f_filtrate_cronologic* (vezi listingul 16.38).

Listingul 16.38. Funcția T-SQL ***f_facturi_filtrate_cronologic***

```
CREATE FUNCTION dbo.f_facturi_filtrate_cronologic
    (@data_initiala SMALLDATETIME, @data_finala SMALLDATETIME)
RETURNS TABLE
AS
RETURN (
    SELECT * FROM facturi WHERE DataFact BETWEEN @data_initiala
                AND @data_finala)
```

Setul de înregistrări furnizat de funcție la invocare poate avea orice structură (atribute). Pe același format sunt redactate și funcțiile *f_facturile_unui_client* (listingul 16.39) și *f_facturi_filtrate* (listingul 16.40).

Listingul 16.39. Funcția T-SQL ***f_facturile_unui_client***

```
CREATE FUNCTION dbo.f_facturile_unui_client
    (@codcl_ SMALLINT)
RETURNS TABLE
    AS
RETURN (
    SELECT * FROM facturi WHERE CodCl = @codcl_)
```

Invocarea acestor funcţii este similară celei din PostgreSQL, cu deosebirea că funcţia-sistem ce furnizează data curentă a sistemului nu este CURRENT_DATE, ci GETDATE():

```
SELECT * FROM f_facturile_unui_client (1001) t
INTERSECT
SELECT *
FROM f_facturi_filtrate_cronologic ('2007-08-05', GETDATE()) t
```

Listingul 16.40. Funcţia T-SQL ***f_facturi_filtrate***

```
CREATE FUNCTION dbo.f_facturi_filtrate
    (@codcl_ SMALLINT, @data_initiala SMALLDATETIME, @data_finala SMALLDATETIME)
RETURNS TABLE
    AS
RETURN (
    SELECT * FROM facturi WHERE CodCl = COALESCE(@codcl_, CodCl)
                AND DataFact BETWEEN COALESCE(@data_initiala, '2005-01-01')
                        AND COALESCE(@data_finala, '2010-12-31')
        )
```

Pentru a pune în valoare flexibilitatea T-SQL în materie de furnizare prin funcţii-utilizator a unui set de înregistrări, redactăm funcţia *f_rute* (listingul 16.41) prin care vom afişa traseele posibile dintre două localităţi trecând prin maximum trei localităţi intermediare. De data aceasta, atributele tabelei returnate (numită *@rute*) se declară chiar în clauza RETURNS. N-am vrut să ne întrebuinţăm prea tare, dar am fi putut declara chiar şi restricţii pentru această tabelă. Popularea sa cu înregistrări se face prin comanda *INSERT INTO @rute VALUES...* . Funcţia se temină printr-o comandă RETURN fără argument.

Listingul 16.41. Funcţia T-SQL ***f_rute***

```
CREATE FUNCTION dbo.f_rute (@plecare VARCHAR(20), @sosire VARCHAR(20))
RETURNS @rute TABLE
(
-- Atributele tabelei returnate
 plecare VARCHAR(20),
 sosire VARCHAR(20),
 ruta VARCHAR(500),
 km NUMERIC(5)
)
BEGIN
 DECLARE @Loc1_1 VARCHAR(20)   DECLARE @Loc2_1 VARCHAR(20)
 DECLARE @Loc1_2 VARCHAR(20)   DECLARE @Loc2_2 VARCHAR(20)
 DECLARE @Loc1_3 VARCHAR(20)   DECLARE @Loc2_3 VARCHAR(20)
 DECLARE @Loc1_4 VARCHAR(20)   DECLARE @Loc2_4 VARCHAR(20)

 DECLARE @ruta1 VARCHAR(500)   DECLARE @ruta2 VARCHAR(500)
 DECLARE @ruta3 VARCHAR(500)   DECLARE @ruta4 VARCHAR(500)

 DECLARE @distanta NUMERIC(5)
 DECLARE @km1 NUMERIC(5)       DECLARE @km2 NUMERIC(5)
 DECLARE @km3 NUMERIC(5)       DECLARE @km4 NUMERIC(5)

 DECLARE c_rute1 CURSOR FOR
SELECT Loc1, Loc2, Distanta FROM distante WHERE Loc1 = @plecare UNION
```

```
        SELECT Loc2, Loc1, Distanta FROM distante WHERE Loc2 = @plecare
  OPEN c_rute1
  FETCH NEXT FROM c_rute1 INTO @Loc1_1, @Loc2_1, @distanta
  WHILE @@FETCH_STATUS = 0 BEGIN
  SET @ruta1 = '*'+ @Loc1_1 + '*' + @Loc2_1 + '*'
  SET @km1 = @distanta
   BEGIN
    IF @Loc2_1 = @sosire
  -- ruta directa
  INSERT INTO @rute VALUES (@Loc1_1, @Loc2_1, @ruta1, @km1)
ELSE
  -- rute printr-o localitate intermediara
BEGIN
  DECLARE c_rute2 CURSOR FOR
    SELECT Loc1, Loc2, Distanta FROM distante
    WHERE Loc1 = @Loc2_1 AND @ruta1 NOT LIKE '%*' + Loc2 + '*%'
       UNION
    SELECT Loc2, Loc1, Distanta FROM distante
    WHERE Loc2 = @Loc2_1 AND @ruta1 NOT LIKE '%*' + Loc1 + '*%'
  OPEN c_rute2
  SET @ruta2 = @ruta1
  FETCH NEXT FROM c_rute2 INTO @Loc1_2, @Loc2_2, @distanta
  WHILE @@FETCH_STATUS = 0 BEGIN
   SET @ruta2 = @ruta1 + @Loc2_2 + '*'
   SET @km2 = @km1 + @distanta
   BEGIN
    IF @Loc2_2 = @sosire
     INSERT INTO @rute VALUES (@plecare, @sosire, @ruta2, @km2)
    ELSE
     -- rute prin doua localitati intermediare
     BEGIN
     DECLARE c_rute3 CURSOR FOR
       SELECT Loc1, Loc2, Distanta FROM distante
       WHERE Loc1 = @Loc2_2 AND @ruta2 NOT LIKE '%*' + Loc2 + '*%'
              UNION
       SELECT Loc2, Loc1, Distanta FROM distante
       WHERE Loc2 = @Loc2_2 AND @ruta2 NOT LIKE '%*' + Loc1 + '*%'
     OPEN c_rute3
     SET @ruta3 = @ruta2
     FETCH NEXT FROM c_rute3 INTO @Loc1_3, @Loc2_3, @distanta
     WHILE @@FETCH_STATUS = 0 BEGIN
      SET @ruta3 = @ruta2 + @Loc2_3 + '*'
      SET @km3 = @km2 + @distanta
      BEGIN
       IF @Loc2_3 = @sosire
        INSERT INTO @rute VALUES (@plecare, @sosire, @ruta3, @km3)
       ELSE
 -- rute prin trei localitati intermediare
        BEGIN
         DECLARE c_rute4 CURSOR FOR
          SELECT Loc1, Loc2, Distanta FROM distante
          WHERE Loc1 = @Loc2_3 AND @ruta3 NOT LIKE '%*' + Loc2 + '*%'
          UNION
          SELECT Loc2, Loc1, Distanta FROM distante
          WHERE Loc2 = @Loc2_3 AND @ruta3 NOT LIKE '%*' + Loc1 + '*%'
         OPEN c_rute4
         SET @ruta4 = @ruta3

         FETCH NEXT FROM c_rute4 INTO @Loc1_4, @Loc2_4, @distanta
```

```
            WHILE @@FETCH_STATUS = 0 BEGIN
          SET @ruta4 = @ruta3 + @Loc2_4 + '*'
          SET @km4 = @km3 + @distanta

          BEGIN
           IF @Loc2_4 = @sosire

           INSERT INTO @rute VALUES (@plecare, @sosire, @ruta4, @km4)
              END

              FETCH NEXT FROM c_rute4 INTO @Loc1_4, @Loc2_4, @distanta
             END
             CLOSE c_rute4

             DEALLOCATE c_rute4
            END
       END
       FETCH NEXT FROM c_rute3 INTO @Loc1_3, @Loc2_3, @distanta
      END
      CLOSE c_rute3
      DEALLOCATE c_rute3
     END
    END
   FETCH NEXT FROM c_rute2 INTO @Loc1_2, @Loc2_2, @distanta
     END
     CLOSE c_rute2
     DEALLOCATE c_rute2
    END
   END
   FETCH NEXT FROM c_rute1 INTO @Loc1_1, @Loc2_1, @distanta
  END
  CLOSE c_rute1
  DEALLOCATE c_rute1
  RETURN
END
```

Iată şi un apel al funcţiei pentru rutele Iaşi-Roman care ne confirmă că funcţia se comportă rezonabil (vezi figura 16.21).

SELECT *

FROM [SQL2008].[dbo].[f_rute] ('Iasi','Roman')

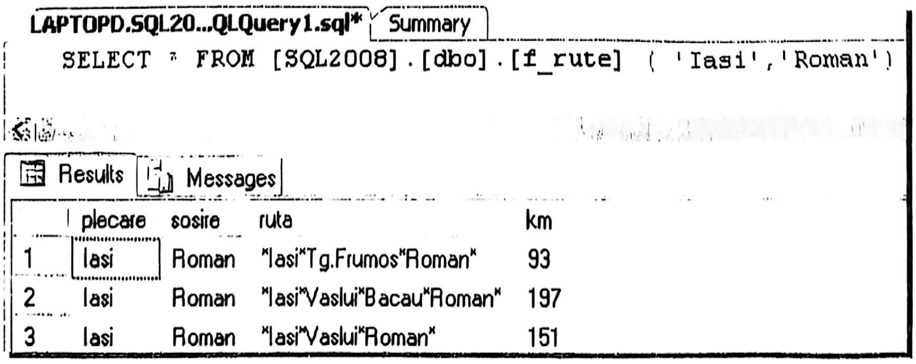


Figura 16.21. Folosirea funcţiei ce returnează o tabelă cu rutele (parţiale) Iaşi-Roman

Acum ne putem întoarce la necazul pricinuit de SQL Server spre finalul paragrafului 12.6 (figura 12.40), când o serie nu putea avea mai mult de 100 de valori

returnate de expresia-tabelă. Lucrul se simplifică acum, pentru că, în locul expresiei-tabelă, redactăm o funcție ce returnează câte o înregistrare pentru orice valoare dintr-un interval. Funcția se numește *f_generare_serii* și este prezentată în listingul 16.42.

Listingul 16.42. Funcția T-SQL ***f_generare_serii***

```
CREATE FUNCTION dbo.f_generare_serii
    (@val_initiala INTEGER, @val_finala INTEGER)
RETURNS @serie TABLE (nr INTEGER )
BEGIN
    DECLARE @i INTEGER
    SET @i = @val_initiala
    WHILE @i <= @val_finala BEGIN
        INSERT INTO @serie VALUES (@i)
        SET @i = @i + 1
    END
    RETURN
END
```

Problema extragerii numerelor de factură nefolosite, rămasă în ceață din paragraful 12.6, presupune execuția interogării următoare care va extrage câte o linie pentru toate numerele cuprinse între valorile minimă și maximă ale numerelor de factură, afișând un mesaj de genul celui din figura 16.22:

nr	NrFact	DataFact	CodCl	Obs	Situatie
1111	1111	2007-08-01 00:00:00	1001	NULL	NULL
1112	1112	2007-08-01 00:00:00	1005	Probleme cu transportul	NULL
1113	1113	2007-08-01 00:00:00	1002	NULL	NULL
1114	1114	2007-08-01 00:00:00	1006	NULL	NULL
1115	1115	2007-08-02 00:00:00	1001	NULL	NULL
1116	1116	2007-08-02 00:00:00	1007	Pretul propus initial a fost modificat	NULL
1117	1117	2007-08-03 00:00:00	1001	NULL	NULL
1118	1118	2007-08-04 00:00:00	1001	NULL	NULL
1119	1119	2007-08-07 00:00:00	1003	NULL	NULL
1120	1120	2007-08-07 00:00:00	1001	NULL	NULL
1121	1121	2007-08-07 00:00:00	1004	NULL	NULL
1122	NULL	NULL	NULL	NULL	Numar nefolosit !
1123	NULL	NULL	NULL	NULL	Numar nefolosit !
1124	NULL	NULL	NULL	NULL	Numar nefolosit !

Figura 16.22. Folosirea funcției ce returnează o serie de numere consecutive

```
SELECT nr, NrFact, DataFact, CodCl, Obs,
        CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !' ELSE NULL END
                AS Situatie
FROM
        (SELECT nr
        FROM dbo.f_generare_serii(1, 9999)
        WHERE nr BETWEEN (SELECT MIN(NrFact) FROM facturi)
```

```
AND (SELECT MAX(NrFact) FROM facturi)
) serie LEFT OUTER JOIN facturi f ON nr = NrFact
```

Singura nemulţumire ar putea fi legată de viteza nu prea mare de execuţie a interogării.

16.3.4. Funcţii ce returnează seturi de înregistrări în SQL PL

N-o să intrăm în detalii despre redactarea funcţiilor SQL PL care furnizează seturi de înregistrări. Ne mulţumim cu listingul 16.43 în care este „portată" funcţia *f_facturi_filtrate*. Spre deosebire de T-SQL, clauza RETURNS TABLE trebuie să conţină numele atributelor tabelei (setului) returnată.

Listingul 16.43. Funcţia SQL PL ***f_facturi_filtrate***

```
CREATE FUNCTION f_facturi_filtrate(codcl_ SMALLINT, data_initiala DATE,
        data_finala DATE )
    RETURNS TABLE(
        NrFact INTEGER, DataFact DATE, CodCl SMALLINT, Obs VARCHAR(50),
                ValTotala DECIMAL (10,2), TVA DECIMAL(10,2)
                        )
    LANGUAGE SQL
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    RETURN   SELECT NrFact, DataFact, CodCl, Obs, ValTotala, TVA
             FROM facturi
             WHERE CodCl = COALESCE(codcl_, CodCl)
                       AND DataFact BETWEEN COALESCE(data_initiala, '2005-01-01')
                       AND COALESCE(data_finala, '2010-12-31');
END
```

Apelul funcţiei în interogări necesită funcţiile TABLE şi CAST. În situaţia în care se doreşte setul facturilor clientului 1001, emise între 7 august 2007 şi data curentă, interogarea va avea sintaxa:

```
SELECT *
FROM TABLE (f_facturi_filtrate( CAST (1001 AS SMALLINT),
        CAST ('2007-08-07' AS DATE), CURRENT_DATE))
```

16.4. Funcţii folosite ca valori implicite şi restricţii la nivel de atribut/înregistrare

O aplicabilitate ceva mai exotică a funcţiilor stocate ţine de asocierea lor unor valori implicite (clauze DEFAULT) şi reguli de validare (clauze CHECK). Numai PostgreSQL şi SQL Server permit lucrul acesta, aşa că doar la acestea ne vom referi în continuare.

Să începem printr-o funcţie PL/pgSQL ce furnizează cel mai frecvent cod de client din facturi (vezi listingul 16.44). Din capitolele anterioare ştim că putem grupa facturile după valorile CodCl şi extrage cea mai mare valoare a numărătorii prin clauza *ORDER BY COUNT(*) DESC LIMIT 1*.

Listingul 16.44. Funcția PL/pgSQL ce urmează să furnizeze valoarea implicită a FACTURI.CodCl

```
CREATE OR REPLACE FUNCTION f_vi_codcl() RETURNS clienti.codcl%TYPE AS
$$
DECLARE
    v_codcl clienti.codcl%TYPE;
BEGIN
    SELECT codcl INTO v_codcl FROM facturi GROUP BY codcl
    ORDER BY COUNT(*) DESC LIMIT 1;

    -- daca nu exista nicio factura introdusa, functia returneaza cel mai mic cod de client
    IF v_codcl IS NULL THEN
        SELECT MIN(CodCl) INTO v_codcl FROM clienti;
    END IF;
    RETURN v_codcl;
END;
$$ LANGUAGE plpgsql;
```

Asocierea acestei funcții cu valoarea implicită a atributului FACTURI.CodCl se realizează prin comanda ALTER TABLE:

ALTER TABLE facturi ALTER COLUMN CodCl SET DEFAULT f_vi_codcl();

Testăm valabilitatea funcției inserând o factură pentru care nu specificăm codul clientului:

INSERT INTO facturi (NrFact, DataFact) VALUES (7050, CURRENT_DATE)

La afişarea acestei linii proaspăt inserate (vezi figura 16.23) observăm că lucrurile au decurs conform aşteptărilor, clientul căruia i-am trimis cele mai multe facturi având codul 1001.

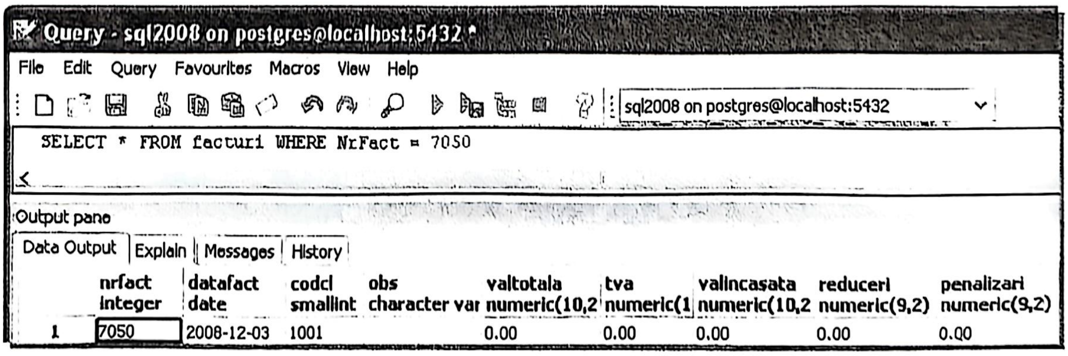


Figura 16.23. Verificarea funcției-valoare implicită

Sintaxa Transact-SQL a aceleiaşi funcții este prezentată în listingul 16.45. Interogarea ce furnizează prima valoare (cea mai frecventă) a atributului *CodCl_* în tabela FACTURI este inclusă într-o comandă SET. La fel şi cea care extrage cel mai mic cod de client.

Listingul 16.45. Funcția T-SQL ce urmează a furniza valoarea implicită a FACTURI.CodCl

```
CREATE FUNCTION dbo.f_vi_codcl ()
    RETURNS SMALLINT
BEGIN
    DECLARE  @codcl SMALLINT
    SET @codcl = 0
```

```
    SET @codcl = (      SELECT TOP 1 CodCl FROM facturi
                        GROUP BY CodCl ORDER BY COUNT(*) DESC
                        )
    -- daca nu exista nicio factura introdusa, functia returneaza cel mai mic cod de client
    IF @codcl = 0
        SELECT @codcl = (SELECT TOP 1 CodCl FROM clienti)

    RETURN @codcl
END
```

Şi sintaxa comenzii ALTER TABLE este un pic diferită în T-SQL:

ALTER TABLE facturi ADD CONSTRAINT def_facturi_codcl

DEFAULT dbo.f_vi_codcl () FOR CodCl

În schimb, testarea funcţiei se face ca în PostgreSQL, doar că INSERT-ul nu poate folosi funcţia CURRENT_DATE, ci GETDATE():

INSERT INTO facturi (NrFact, DataFact) VALUES (7050, getdate())

Continuăm cu o funcţie pe care vrem să o asociem unei clauze CHECK – regulă de validare la nivel de linie. Probabil că în capitolele 3 şi 13 aţi fost dezamăgiţi de sărăcia clauzei CHECK a comenzilor CREATE/ALTER TABLE. În aplicaţiile economice, deseori trebuie gestionate restricţii mult mai complexe decât simplele expresii permise de clauza CHECK. Chiar dacă soluţia generală de implementare a (aproape) tuturor restricţiilor o reprezintă declanşatoarele (*trigger*-ele) discutate în capitolul următor, PostgreSQL şi SQL Server acceptă ca, în clauzele CKECK, în locul unui predicat „clasic" să fie specificat numele unei funcţii.

Să luăm cazul unei funcţii ce primeşte ca parametru de intrare valoarea codului unui client (*CodCl_*) şi furnizează TRUE sau FALSE dacă restul de plată al acelui client, adică diferenţa dintre vânzări şi încasări, este mai mic sau mai mare decât 175000000 RON. Funcţia se numeşte *f_ck_facturi_codcl* şi este prezentată în listingul 16.46.

Listingul 16.46. Funcţia PL/pgSQL ce urmează a fi clauză CHECK în tabela FACTURI

```
CREATE OR REPLACE FUNCTION f_ck_facturi_codcl (codcl_ facturi.CodCl%TYPE)
    RETURNS BOOLEAN AS
$$
BEGIN
    RETURN CASE WHEN
    COALESCE((
                SELECT SUM(Cantitate * PretUnit * (1 + f_procTVA (lf.CodPr)))
                FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                WHERE CodCl = codcl_),0) -
                        COALESCE((
                                SELECT SUM(Transa)
                                FROM facturi f INNER JOIN incasfact i
                                        ON f.NrFact = i.NrFact
                                WHERE CodCl = codcl_
                                ),0) < 175000000 THEN TRUE ELSE FALSE
        END;
END;
$$ LANGUAGE plpgsql;
```

Declararea sa ca regulă de validare pentru tabela FACTURI se realizează, firesc, printr-o comandă ALTER TABLE:

ALTER TABLE facturi ADD CONSTRAINT ck_facturi_codcl

CHECK (f_ck_facturi_codcl (codcl));

Nu vă rămâne decât să testaţi funcţionalitatea noii reguli, introducând o factură cu o valoare foarte mare. Versiunea Transact-SQL a funcţiei *f_ck_facturi_codcl* reprezintă subiectul listingului 16.47.

Listingul 16.47. Funcţia T-SQL ce urmează a fi clauză CHECK în tabela FACTURI

```
CREATE FUNCTION dbo.f_ck_facturi_codcl (@codcl SMALLINT)
    RETURNS BIT
BEGIN
    RETURN CASE WHEN
    COALESCE((
                SELECT SUM(Cantitate * PretUnit * (1 + dbo.f_proctva (lf.CodPr)))
                FROM facturi f INNER JOIN liniifact lf ON f.NrFact = lf.NrFact
                WHERE CodCl = @codcl),0) -
                        COALESCE((
                                SELECT SUM(Transa)
                                FROM facturi f INNER JOIN incasfact i
                                        ON f.NrFact = i.NrFact
                                WHERE CodCl = @codcl
                                ),0) < 175000000 THEN 0 ELSE 1
        END
END
```

Comanda ALTER TABLE pentru asocierea acestei funcţii unei clauze CHECK a tabelei FACTURI are sintaxa SQL Server:

ALTER TABLE facturi WITH CHECK ADD CONSTRAINT ck_facturi_codcl

CHECK (dbo.f_ck_facturi_codcl (CodCl) = 0);

16.5. Proceduri stocate

Mulţi dintre noi ştiu din tinereţe (măcar) că, dacă o funcţie returnează „ceva", o procedură face „ceva". Definiţia nu este dintre cele mai ştiinţifice, dar are calităţile ei. Faţă de funcţii, discuţia legată de proceduri este mai sumară. Dintre cele patru servere BD, PostgreSQL-ul nu are proceduri propriu-zise, ci doar funcţii care returnează VOID.

16.5.1. Proceduri stocate în PostgreSQL

Începem cu o funcţie-procedură care adaugă în observaţiile (Obs) fiecărei facturi mesajul *Nu are linii!* dacă numărul de factură respectiv nu se regăseşte în tabela LINIFACT (vezi listingul 16.48).

Listingul 16.48. Funcţia-procedură SQL ce „actualizează" valorile atributului Obs

```
CREATE OR REPLACE FUNCTION f_update_facturi_fara_linii()
    RETURNS VOID AS $$
        UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii!'
        WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
$$ LANGUAGE SQL;
```

Apelarea procedurii înseamnă, de fapt, apelarea funcţiei, în maniera din paragraful 16.1.1:

SELECT f_update_facturi_fara_linii()

Acestei funcţii-proceduri i se poate reproşa că, la executări repetate, umple observaţiile cu mesaje „Nu are linii!", aşa că ne putem gândi la o varianta mai acceptabilă (vezi listingul 16.49).

Listingul 16.49. O variantă mai răsărită a funcţiei-procedură

```
CREATE OR REPLACE FUNCTION f_update_facturi_fara_linii()
    RETURNS VOID AS $$
        UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
        WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
                AND COALESCE(obs,' ') NOT LIKE '%Nu are linii !%'
$$ LANGUAGE SQL;
```

Continuăm cu o funcţie care se substituie clauzei ON DELETE CASCADE din comenzi CREATE/ALTER TABLE. La drept vorbind, aveam nevoie de o asemenea procedură doar în DB2 şi Oracle, însă o vom redacta şi în celelalte două servere BD. Listingul 16.50 conţine funcţia *f_on_delete_cascade_factura* dedicată ştergerii unei facturi şi tuturor înregistrărilor-copil ale acesteia:

Listingul 16.50. O funcţie-procedură de tip ON DELETE CASCADE

```
CREATE OR REPLACE FUNCTION f_on_delete_cascade_factura (nrfact_ facturi.nrfact%TYPE)
        RETURNS VOID AS $$
    DELETE FROM incasfact WHERE nrfact = $1;
    DELETE FROM liniifact WHERE nrfact = $1;
    DELETE FROM facturi WHERE nrfact = $1;
$$ LANGUAGE SQL;
```

Continuăm cu o procedură dedicată admiterii la masterat, discutată în paragraful 13.3.2. Mai întâi creăm cele două tabele, MASTERE şi CANDIDAŢI (vezi listingul 16.51).

Listingul 16.51. Crearea (PostgreSQL) celor două tabele pentru admiterea la masterat

```
CREATE TABLE mastere (
    Spec VARCHAR(4) PRIMARY KEY,
    DenSpec VARCHAR(40) NOT NULL,
    NrLocuri NUMERIC(3),
    NrLocuri_Ocupate NUMERIC(3)
    );

CREATE TABLE candidati (
    IdCandidat NUMERIC(6) NOT NULL PRIMARY KEY,
    NumePren VARCHAR(50) NOT NULL,
    Media_AS NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_as
        CHECK (media_as BETWEEN 5 AND 10),
    Media_Lic NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_lic
        CHECK (media_lic BETWEEN 6 AND 10),
    Spec1 VARCHAR(4) NOT NULL REFERENCES mastere(Spec),
    Spec2 VARCHAR(4) REFERENCES mastere(Spec),
    Spec3 VARCHAR(4) REFERENCES mastere(Spec),
    Spec4 VARCHAR(4) REFERENCES mastere(Spec),
    Spec5 VARCHAR(4) REFERENCES mastere(Spec),
    Spec6 VARCHAR(4) REFERENCES mastere(Spec),
```

```
        Spec_Repartizat VARCHAR(4) REFERENCES mastere(Spec),
        CONSTRAINT ck_spec CHECK (
            COALESCE(spec2, ' ') = COALESCE(spec2, spec3, ' ') AND
            COALESCE(spec3, ' ') = COALESCE(spec3, spec4, ' ') AND
            COALESCE(spec4, ' ') = COALESCE(spec4, spec5, ' ') AND
            COALESCE(spec5, ' ') = COALESCE(spec5, spec6, ' '))
                    );
```

Urmează popularea celor două tabele (vezi listingul 16.52).

Listingul 16.52. Popularea (PostgreSQL) tabelelor MASTERE și CANDIDAȚI

```
DELETE FROM candidati;
DELETE FROM mastere;

INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('SIA',
    'Sisteme informationale pentru afaceri', 4);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('FAB', 'Finante-Asigurari-Banci', 6);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('CEA',
    'Contabilitate, expertiza si audit', 6);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MARK', 'Marketing', 5);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('EAI',
    'Economie si afaceri internationale', 5);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MRU',
    'Managementul resurselor umane', 4);

INSERT INTO candidati VALUES ( 1, 'Popescu I. Irina', 7.5, 9.50, 'FAB', 'MARK', 'EAI', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 2, 'Babias D. Ecaterina', 8.5, 9.20, 'SIA', 'EAI', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 3, 'Strat P. Iulian', 7.35, 9.50, 'CEA', 'EAI', 'MRU', 'SIA',
    'FAB', 'MARK', NULL);
INSERT INTO candidati VALUES ( 4, 'Georgescu M. Honda', 8.5, 9.00, 'MRU', 'MARK', 'EAI',
    'FAB', 'CEA', 'SIA', NULL);
INSERT INTO candidati VALUES ( 5, 'Munteanu A. Optimista', 9.5, 9.50, 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 6, 'Dumitriu F. Dura', 9.5, 10, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 7, 'Mesnita G. Plinutul', 10, 10, 'EAI', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 8, 'Greavu V. Doru', 9.25, 8.50, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 9, 'Baboi P. Iustina', 8.5, 8.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (10, 'Postelnicu I. Irina', 9.5, 8.25, 'MARK', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (11, 'Fotache H. Fanel', 9.75, 9.50, 'MRU', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (12, 'Moscovici J. Cristina', 9.80, 9.30, 'CEA', 'SIA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (13, 'Rusu I. Vanda', 7.80, 9.10, 'FAB', 'CEA', 'MARK', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (14, 'Spinu M. Algebra', 7.25, 9.00, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (15, 'Sandovici I. Irina', 7.05, 7.50, 'MARK', 'MRU', 'EAI', 'CEA',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (16, 'Plai V. Picior', 7.5, 7.90, 'EAI', 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (17, 'Ambuscada B. Cristina', 8.25, 9.50, 'SIA', 'CEA',
    'FAB', 'EAI', NULL, NULL, NULL);
```

```
INSERT INTO candidati VALUES (18, 'Pinda A. Axinia', 8.75, 9.00, 'SIA', 'FAB', 'CEA', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (19, 'Planton V. Grigore', 9.25, 9.50, 'FAB', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (20, 'Sergentu I. Zece',7.5, 9.50, 'FAB', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (21, 'Ababei T. Marian-Vasile', 7.5, 9.50, 'CEA', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (22, 'Popistasu J. Maria', 7.5, 9.50, 'FAB', 'EAI', 'CEA', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (23, 'Plop R. Robert', 7.5, 9.50, 'FAB', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (24, 'Aflorei H. Crina',7.5, 9.50, 'EAI', 'SIA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (25, 'Afaunei P. Gina',7.5, 9.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (26, 'Vatamanu I. Alexandrina', 7.5, 9.50, 'MRU', 'MARK', 'EAI',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (27, 'Lovin P. Marian',7.5, 9.50, 'MARK', 'MRU', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (28, 'Antiteza W. Florin', 7.5, 9.50, 'EAI', 'MARK', 'MRU',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (29, 'Prepelita V. Ion', 7.5, 9.50, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (30, 'Cioara X. Sanda', 7.5, 9.50, 'CEA', 'FAB', 'EAI', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (31, 'Metafora Y. Vasile',7.5, 9.50, 'SIA', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (32, 'Strasina R. Elvis', 7.5, 9.50, 'CEA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (33, 'Durere V. Vasile', 7.5, 9.50, 'FAB', 'CEA', 'EAI', 'MARK',
    'MRU', NULL, NULL);
INSERT INTO candidati VALUES (34, 'Sedentaru L. Marius-Daniel',7.5, 9.50, 'MARK', 'MRU',
    'EAI', NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (35, 'Zgircitu I. Daniel',7.5, 9.50, 'MRU', NULL, NULL, NULL,
    NULL, NULL, NULL);
```

Funcţia-procedură *p_admitere* (listingul 16.53) aduce câteva noutăţi PL/pgSQL. Cursorul *c_mastere* este unul parametrizat, la deschiderea sa făcându-se filtrarea în funcţie de valoarea curentă a parametrului *CodCl_*. Ambele cursoare prezintă opţiunea FOR UPDATE, ceea ce înseamnă că, pe baza înregistrării curente a cursorului, se va putea face actualizarea liniei din tabela de bază din care provine cursorul. Din păcate, în PL/pgSQL, lucrurile funcţionează numai pentru cursorul *c_mastere* (nu ştiu de ce pentru *c_candidaţi* lucrurile nu sunt în regulă la folosirea clauzei FOR UPDATE). Există, de asemenea, trei variabile de tip RECORD care este un tip linie (înregistrare) general.

Listingul 16.53. Procedura PL/pgSQL pentru admiterea candidaţilor

```
CREATE OR REPLACE FUNCTION p_admitere () RETURNS VOID AS
$$
DECLARE
    DECLARE c_mastere CURSOR (spec_ mastere.spec%TYPE) IS
        SELECT * FROM mastere WHERE spec = spec_ AND NrLocuri > NrLocuri_Ocupate
        FOR UPDATE;
    DECLARE c_candidati CURSOR IS
        SELECT * FROM candidati ORDER BY Media_AS * 0.6 + Media_Lic * 0.4 DESC
```

```
        FOR UPDATE ;
    rec_candidati RECORD;
    rec_optiuni RECORD;
    rec_mastere RECORD;

BEGIN
    UPDATE mastere SET NrLocuri_Ocupate = 0;
    UPDATE candidati SET spec_repartizat = NULL;

    OPEN c_candidati;
    LOOP
        FETCH c_candidati INTO rec_candidati;
        IF NOT FOUND THEN
                EXIT;
        END IF;

        FOR rec_optiuni IN (
                SELECT rec_candidati.IdCandidat, 1 AS OptiuneNr, rec_candidati.Spec1 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 2 AS OptiuneNr, rec_candidati.Spec2 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 3 AS OptiuneNr, rec_candidati.Spec3 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 4 AS OptiuneNr, rec_candidati.Spec4 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 5 AS OptiuneNr, rec_candidati.Spec5 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 6 AS OptiuneNr, rec_candidati.Spec6 AS Spec
                                        ORDER BY 2 ) LOOP

                IF rec_optiuni.Spec IS NULL THEN
                        EXIT;
                END IF;

                OPEN c_mastere (rec_optiuni.Spec);
                FETCH c_mastere INTO rec_mastere;
                IF FOUND THEN
                        UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                        WHERE CURRENT OF c_mastere;
                        UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                        WHERE IdCandidat = rec_candidati.IdCandidat;
                                -- nu functioneaza CURRENT OF c_candidati!!!!!
                        CLOSE c_mastere;
                        EXIT;
                ELSE
                        CLOSE c_mastere;
                END IF;
        END LOOP ;
    END LOOP;
    CLOSE c_candidati;
END;
$$ LANGUAGE plpgsql;
```

Funcția-procedură începe prin a face curățenie, NULL-izând atributul *spec_repartizat* pentru toți candidații și „zerorizând" valorile atributului *NrLocuri_Ocupate* pentru toate specializările de mastert. Apoi se parcurg candidații în ordinea descrescătoare a mediei de admitere (expresia *Media_AS * 0.6 + Media_Lic * 0.4*). Pentru fiecare candidat se construiește un cursor ad-hoc ale cărui

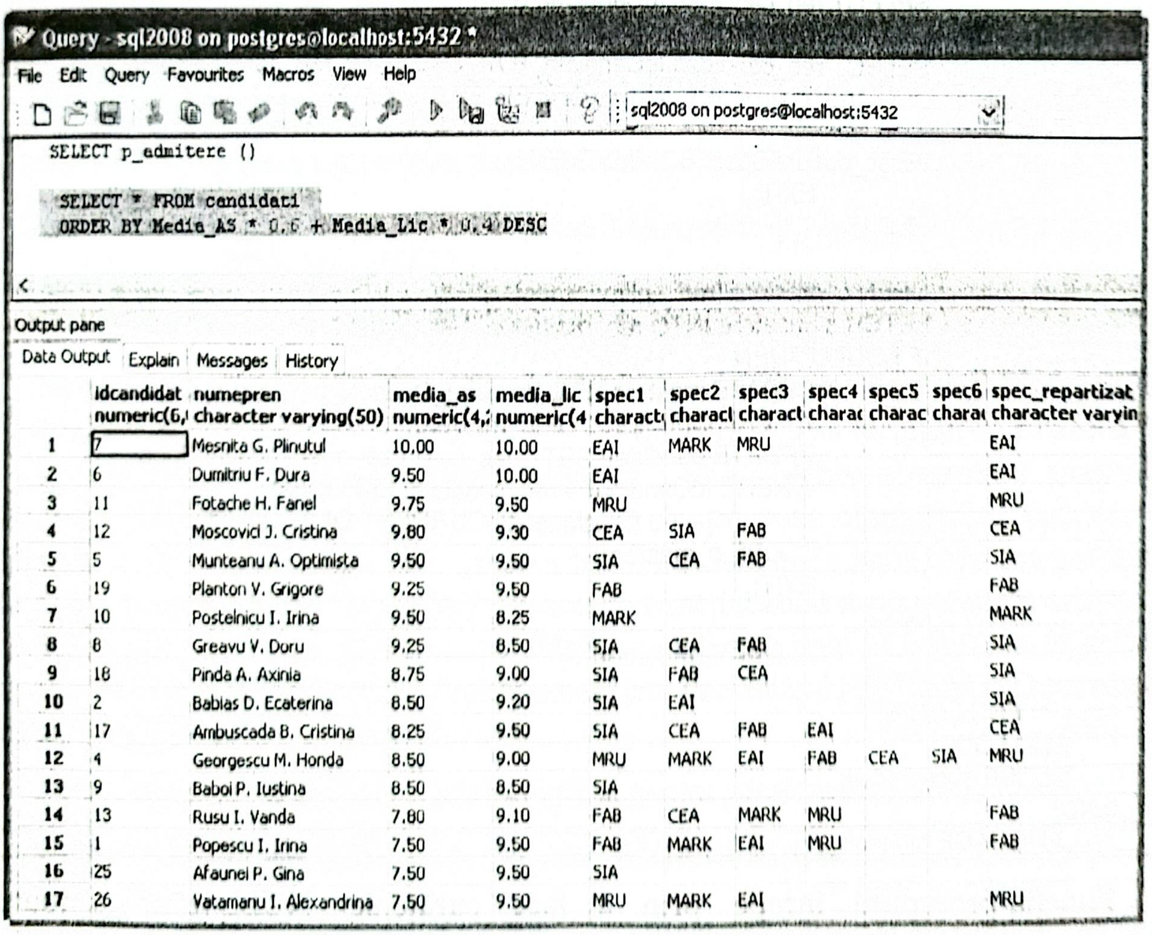
înregistrări conţin opţiunile pe care le-a bifat candidatul (ordinea opţiunilor este esenţială). Se parcurg toate aceste opţiuni până când, fie candidatului i se găseşte o specializare la care mai sunt locuri neocupate, fie opţiunea următoare este nulă (ceea ce înseamnă că pe candidat nu l-au interesat toate cele şase specializări). Dacă i se găseşte o specializare (dorită) în care există locuri disponibile, atunci pentru acest candidat atributul *spec_repartizat* preia numele specializării, iar acesteia i se incrementează numărul locurilor ocupate.

Lansarea în execuţie este simplă:

SELECT p_admitere ()

După lansarea procedurii, examinăm o parte din conţinutul tabelei CANDIDAŢI ordonat după medie (vezi figura 16.24) şi observăm că, pe linia 11, candidata *Ambuscadă B. Cristina* a avut ca primă opţiune specializarea *SIA* (Sisteme informaţionale pentru afaceri), însă a fost repartizată la *CEA* (Contabilitate, Expertiză şi Audit). Motivul este că la SIA au existat doar patru locuri care au fost ocupate de alţi candidaţi cu medie mai mare.

În figură, două candidate, Baboi P. Iustina şi Afaunei P. Gina sunt respinse (valoarea atributului *spec_repartizat* este NULL), întrucât acestea au dorit numai această specializare ale cărei locuri au fost ocupate de candidaţii cu medii mai mari.




```
SELECT p_admitere ()

SELECT * FROM candidati
ORDER BY Media_AS * 0.6 + Media_Lic * 0.4 DESC
```

	idcandidat numeric(6,(	numepren character varying(50)	media_as numeric(4,2	media_lic numeric(4	spec1 charact	spec2 charact	spec3 charact	spec4 charac	spec5 charac	spec6 charac	spec_repartizat character varyin
1	7	Mesnita G. Plinutul	10.00	10.00	EAI	MARK	MRU				EAI
2	6	Dumitriu F. Dura	9.50	10.00	EAI						EAI
3	11	Fotache H. Fanel	9.75	9.50	MRU						MRU
4	12	Moscovici J. Cristina	9.80	9.30	CEA	SIA	FAB				CEA
5	5	Munteanu A. Optimista	9.50	9.50	SIA	CEA	FAB				SIA
6	19	Planton V. Grigore	9.25	9.50	FAB						FAB
7	10	Postelnicu I. Irina	9.50	8.25	MARK						MARK
8	8	Greavu V. Doru	9.25	8.50	SIA	CEA	FAB				SIA
9	18	Pinda A. Axinia	8.75	9.00	SIA	FAB	CEA				SIA
10	2	Babias D. Ecaterina	8.50	9.20	SIA	EAI					SIA
11	17	Ambuscada B. Cristina	8.25	9.50	SIA	CEA	FAB	EAI			CEA
12	4	Georgescu M. Honda	8.50	9.00	MRU	MARK	EAI	FAB	CEA	SIA	MRU
13	9	Baboi P. Iustina	8.50	8.50	SIA						
14	13	Rusu I. Vanda	7.80	9.10	FAB	CEA	MARK	MRU			FAB
15	1	Popescu I. Irina	7.50	9.50	FAB	MARK	EAI	MRU			FAB
16	25	Afaunei P. Gina	7.50	9.50	SIA						
17	26	Vatamanu I. Alexandrina	7.50	9.50	MRU	MARK	EAI				MRU

Figura 16.24. Verificarea funcţiei-procedură ***p_admitere***

16.5.2. Proceduri stocate în Oracle PL/SQL

Sintaxa și funcționalitatea procedurilor stocate redactate în Oracle PL/SQL respectă canoanele „clasice". Începem cu o procedură care afișează în dreptul fiecărei facturi atât valoarea sa, cât și valoarea cumulată a facturilor precedente (plus a celei curente - vezi listingul 16.54).

Listingul 16.54. Procedura PL/SQL ***p_cumul***

```
-- procedura de afisare a valorii facturii curente si a sumei valorilor facturilor precedente
-- varianta 2 - cursor
CREATE OR REPLACE PROCEDURE p_cumul2
IS
    v_suma NUMBER(16,2) := 0;
BEGIN
    FOR rec_fact IN (
                SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
                FROM facturi) LOOP
        v_suma := v_suma + NVL(rec_fact.valfact,0);
        DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                TO_CHAR(v_suma, '999999999999.99') );
    END LOOP;
END;
```

Procedura care adaugă la observațiile unei facturi un eventual mesaj care semnalizează că factura nu are nicio linie (echivalentă celei din listingul 16.49) este subiectul listingului 16.55.

Listingul 16.55. Procedura PL/SQL ***p_update_facturi_fara_linii***

```
CREATE OR REPLACE PROCEDURE p_update_facturi_fara_linii
IS
BEGIN
    UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
    WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
        AND COALESCE(obs,' ') NOT LIKE '%Nu are linii !%';
END;
/
```

Lansarea procedurii se poate face dintr-un bloc anonim astfel:

```
BEGIN
p_update_facturi_fara_linii;
END;
```

Continuăm cu funcția-substitut a clauzei ON DELETE CASCADE (vezi listingul 16.56), mult mai utilă în Oracle decât în PostgreSQL și SQL Server.

Listingul 16.56. Procedura PL/SQL ***p_on_delete_cascade_factura***

```
CREATE OR REPLACE PROCEDURE
    p_on_delete_cascade_factura (nrfact_ facturi.nrfact%TYPE)
IS
BEGIN
    DELETE FROM incasfact WHERE nrfact = nrfact_;
    DELETE FROM liniifact WHERE nrfact = nrfact_;
    DELETE FROM facturi WHERE nrfact = nrfact_;
END;
```

Ajungem la procedura dedicată admiterii la master. În Oracle avem deja create tabelele (din paragraful 13.3.2), însă la structura inițială trebuie să mai facem câteva adăugiri:

ALTER TABLE mastere ADD (NrLocuri_Ocupate NUMERIC(3) DEFAULT 0);

ALTER TABLE candidati ADD (Spec_Repartizat VARCHAR2(4)

REFERENCES mastere(Spec));

Procedura seamănă leit cu cea din PL/pgSQL (din listingul 16.53) (vezi listingul 16.57) cu observația că ambele cursoare pot actualiza tabelele prin comanda *UPDATE... FOR CURRENT...* .

Listingul 16.57. Procedura PL/SQL ***p_admitere***

```
CREATE OR REPLACE PROCEDURE p_admitere
AS
    CURSOR c_mastere (spec_ mastere.spec%TYPE) IS
        SELECT * FROM mastere WHERE spec = spec_ AND NrLocuri > NrLocuri_Ocupate
                FOR UPDATE ;
    CURSOR c_candidati IS
        SELECT * FROM candidati ORDER BY (Media_AS * 0.6 + Media_Lic * 0.4) DESC
                FOR UPDATE ;
BEGIN
    UPDATE mastere SET NrLocuri_Ocupate = 0;
    UPDATE candidati SET spec_repartizat = NULL;

    FOR rec_candidati IN c_candidati LOOP
        FOR rec_optiuni IN (
                SELECT rec_candidati.IdCandidat, 1 AS OptiuneNr, rec_candidati.Spec1 AS Spec
                          FROM dual UNION
                SELECT rec_candidati.IdCandidat, 2 AS OptiuneNr, rec_candidati.Spec2 AS Spec
                          FROM dual UNION
                SELECT rec_candidati.IdCandidat, 3 AS OptiuneNr, rec_candidati.Spec3 AS Spec
                          FROM dual UNION
                SELECT rec_candidati.IdCandidat, 4 AS OptiuneNr, rec_candidati.Spec4 AS Spec
                          FROM dual UNION
                SELECT rec_candidati.IdCandidat, 5 AS OptiuneNr, rec_candidati.Spec5 AS Spec
                          FROM dual UNION
                SELECT rec_candidati.IdCandidat, 6 AS OptiuneNr, rec_candidati.Spec6 AS Spec
                          FROM dual ORDER BY 2 ) LOOP
                IF rec_optiuni.Spec IS NULL THEN
                          EXIT;
                END IF;

                FOR rec_mastere IN c_mastere (rec_optiuni.Spec) LOOP
                          UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                                    WHERE CURRENT OF c_mastere;
                          UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                                    WHERE CURRENT OF c_candidati;
                          EXIT;
                END LOOP ;
        END LOOP;
    END LOOP;
END;
```

Încheiem paragraful cu o procedură destinată actualizării tabelei LINIIFACT - astfel încât liniile din fiecare factură să fie consecutive. Cu alte cuvinte, dacă dintr-o factură se şterge o înregistrare (să zicem, linia 2) care nu este ultima, ar apărea un

interval (gaură) în numerotarea liniilor. Or, procedura *p_renumerotare_linii* (din listingul 16.58) „astupă găurile" din toată tabela.

Listingul 16.58. Procedura PL/SQL ***p_renumerotare_linii***

```
CREATE OR REPLACE PROCEDURE p_renumerotare_linii
AS i NUMBER(2);
BEGIN
 FOR rec_facturi IN (SELECT NrFact FROM liniifact
   GROUP BY NrFact HAVING COUNT(*) <> MAX(Linie)
            ) LOOP
   i := 1;
   FOR rec_liniifact IN (SELECT * FROM liniifact WHERE NrFact = rec_facturi.NrFact
                                   ORDER BY Linie) LOOP
    IF rec_liniifact.linie <> i THEN
      UPDATE liniifact SET linie = 1
               WHERE NrFact = rec_facturi.NrFact AND linie = rec_liniifact.linie;
    END IF;
    i := i + 1;
   END LOOP;
 END LOOP;
END;
```

16.5.3. Proceduri stocate în Transact-SQL

Procedurile stocate în T-SQL au câteva note de exotism. Mai întâi, ca și în PostgreSQL (deși în PostgreSQL nu avem proceduri), într-o procedură T-SQL putem înșirui oricâte comenzi SQL, inclusiv SELECT-uri. Prin urmare, procedurile pot semăna pe alocuri cu scripturile. În altă ordine de idei, analog tabelelor, procedurile pot fi declarate ca fiind temporare, locale sau globale. Noi ne vom feri de exotisme, păstrând exemplele din paragrafele anterioare. Începem cu procedura *p_update_facturi_fără_linii* din listingul 16.59.

Listingul 16.59. Procedura T-SQL ***p_update_facturi_fără_linii()***

```
CREATE PROCEDURE p_update_facturi_fara_linii
AS
BEGIN
    UPDATE facturi SET Obs = COALESCE(Obs,' ') + 'Nu are linii!'
    WHERE NrFact NOT IN (SELECT DISTINCT NrFact FROM liniifact)
                AND COALESCE(Obs,' ') NOT LIKE '%Nu are linii!%'
END
```

Apelul procedurii se face prin comanda EXECUTE:

EXECUTE p_update_facturi_fara_linii

Continuăm cu procedura dedicată admiterii la masterat. Listingul 16.60 conține comenzile de creare și populare a celor două tabele.

Listingul 16.60. Crearea/popularea în T-SQL a celor două tabele pentru admiterea la masterat

```
CREATE TABLE mastere (
    Spec VARCHAR(4) NOT NULL PRIMARY KEY,
    DenSpec VARCHAR(40) NOT NULL,
    NrLocuri NUMERIC(3),
```

```
    NrLocuri  Ocupate NUMERIC(3)
    );

CREATE TABLE candidati (
    IdCandidat NUMERIC(6) NOT NULL PRIMARY KEY,
    NumePren VARCHAR(50) NOT NULL,
    Media_AS NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_as
        CHECK (Media_AS BETWEEN 5 AND 10),
    Media_Lic NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_lic
        CHECK (Media_Lic BETWEEN 6 AND 10),
    Spec1 VARCHAR(4) NOT NULL REFERENCES mastere(Spec),
    Spec2 VARCHAR(4) REFERENCES mastere(Spec),
    Spec3 VARCHAR(4) REFERENCES mastere(Spec),
    Spec4 VARCHAR(4) REFERENCES mastere(Spec),
    Spec5 VARCHAR(4) REFERENCES mastere(Spec),
    Spec6 VARCHAR(4) REFERENCES mastere(Spec),
    Spec_Repartizat VARCHAR(4) REFERENCES mastere(Spec),
    CONSTRAINT ck_spec CHECK (
        COALESCE(Spec2, ' ') = COALESCE(Spec2, Spec3, ' ') AND
        COALESCE(Spec3, ' ') = COALESCE(Spec3, Spec4, ' ') AND
        COALESCE(Spec4, ' ') = COALESCE(Spec4, Spec5, ' ') AND
        COALESCE(Spec5, ' ') = COALESCE(Spec5, Spec6, ' ')   )
            );

INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('SIA',
    'Sisteme informationale pentru afaceri', 4);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('FAB', 'Finante-Asigurari-Banci', 6);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('CEA',
    'Contabilitate, expertiza si audit', 6);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MARK', 'Marketing', 5);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('EAI',
    'Economie si afaceri internationale', 5);
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MRU',
    'Managementul resurselor umane', 4);

INSERT INTO candidati VALUES ( 1, 'Popescu I. Irina', 7.5, 9.50, 'FAB', 'MARK', 'EAI', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 2, 'Babias D. Ecaterina', 8.5, 9.20, 'SIA', 'EAI', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 3, 'Strat P. Iulian', 7.35, 9.50, 'CEA', 'EAI', 'MRU', 'SIA',
    'FAB', 'MARK', NULL);
INSERT INTO candidati VALUES ( 4, 'Georgescu M. Honda', 8.5, 9.00, 'MRU', 'MARK', 'EAI',
    'FAB', 'CEA', 'SIA', NULL);
INSERT INTO candidati VALUES ( 5, 'Munteanu A. Optimista', 9.5, 9.50, 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 6, 'Dumitriu F. Dura', 9.5, 10, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 7, 'Mesnita G. Plinutul', 10, 10, 'EAI', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 8, 'Greavu V. Doru', 9.25, 8.50, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 9, 'Baboi P. Iustina', 8.5, 8.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (10, 'Postelnicu I. Irina', 9.5, 8.25, 'MARK', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (11, 'Fotache H. Fanel', 9.75, 9.50, 'MRU', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (12, 'Moscovici J. Cristina', 9.80, 9.30, 'CEA', 'SIA', 'FAB',
    NULL, NULL, NULL, NULL);
```

```
INSERT INTO candidati VALUES (13, 'Rusu I. Vanda', 7.80, 9.10, 'FAB', 'CEA', 'MARK', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (14, 'Spinu M. Algebra', 7.25, 9.00, 'SIA', 'CEA', 'FAB', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (15, 'Sandovici I. Irina', 7.05, 7.50, 'MARK', 'MRU', 'EAI', 'CEA',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (16, 'Plai V. Picior', 7.5, 7.90, 'EAI', 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (17, 'Ambuscada B. Cristina', 8.25, 9.50, 'SIA', 'CEA',
    'FAB', 'EAI', NULL, NULL, NULL);
INSERT INTO candidati VALUES (18, 'Pinda A. Axinia', 8.75, 9.00, 'SIA', 'FAB', 'CEA', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (19, 'Planton V. Grigore', 9.25, 9.50, 'FAB', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (20, 'Sergentu I. Zece',7.5, 9.50, 'FAB', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (21, 'Ababei T. Marian-Vasile', 7.5, 9.50, 'CEA', 'MRU',
    NULL, NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (22, 'Popistasu J. Maria', 7.5, 9.50, 'FAB', 'EAI', 'CEA', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (23, 'Plop R. Robert', 7.5, 9.50, 'FAB', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (24, 'Aflorei H. Crina',7.5, 9.50, 'EAI', 'SIA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (25, 'Afaunei P. Gina',7.5, 9.50, 'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (26, 'Vatamanu I. Alexandrina', 7.5, 9.50, 'MRU', 'MARK', 'EAI',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (27, 'Lovin P. Marian',7.5, 9.50, 'MARK', 'MRU', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (28, 'Antiteza W. Florin', 7.5, 9.50, 'EAI', 'MARK', 'MRU',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (29, 'Prepelita V. Ion', 7.5, 9.50, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (30, 'Cioara X. Sanda', 7.5, 9.50, 'CEA', 'FAB', 'EAI', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (31, 'Metafora Y. Vasile',7.5, 9.50, 'SIA', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (32, 'Strasina R. Elvis', 7.5, 9.50, 'CEA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (33, 'Durere V. Vasile', 7.5, 9.50, 'FAB', 'CEA', 'EAI', 'MARK',
    'MRU', NULL, NULL);
INSERT INTO candidati VALUES (34, 'Sedentaru L. Marius-Daniel',7.5, 9.50, 'MARK', 'MRU',
    'EAI', NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (35, 'Zgircitu I. Daniel',7.5, 9.50, 'MRU', NULL, NULL, NULL,
    NULL, NULL, NULL);
```

Procedura *p_admitere* (listingul 16.61) folosește, ca și suratele ei din PL/pgSQL și PL/SQL, două cursoare, *c_candidați* și *c_opțiuni*. Pentru ambele cursoare se folosește (și funcționează) clauzele *FOR UPDATE ...* și *FOR CURRENT OF...* .

Listingul 16.61. Procedura T-SQL pentru admiterea candidaților

```
CREATE PROCEDURE p_admitere AS
BEGIN
    DECLARE @spec VARCHAR(4)
    DECLARE @gata BIT
    DECLARE @nr TINYINT
```

```
DECLARE @idcandidat NUMERIC(6)
DECLARE @spec1 VARCHAR(4)          DECLARE @spec2 VARCHAR(4)
DECLARE @spec3 VARCHAR(4)          DECLARE @spec4 VARCHAR(4)
DECLARE @spec5 VARCHAR(4)          DECLARE @spec6 VARCHAR(4)

DECLARE @spec_m VARCHAR(4) DECLARE @denspec VARCHAR(40)
DECLARE @nrlocuri NUMERIC(3) DECLARE @nrlocuri_ocupate NUMERIC(3)

UPDATE mastere SET NrLocuri_Ocupate = 0
UPDATE candidati SET Spec_Repartizat = NULL

DECLARE c_candidati CURSOR FOR
    SELECT IdCandidat, Spec1, Spec2, Spec3, Spec4, Spec5, Spec6
    FROM candidati ORDER BY Media_AS * 0.6 + Media_Lic * 0.4 DESC FOR UPDATE

OPEN c_candidati;
FETCH NEXT FROM c_candidati INTO @idcandidat, @spec1, @spec2, @spec3,
    @spec4, @spec5, @spec6
WHILE @@FETCH_STATUS = 0 BEGIN

    DECLARE c_optiuni CURSOR FOR
            SELECT 1 AS Nr, @spec1 AS Spec UNION
            SELECT 2 AS Nr, @spec2 AS Spec UNION
            SELECT 3 AS Nr, @spec3 AS Spec UNION
            SELECT 4 AS Nr, @spec4 AS Spec UNION
            SELECT 5 AS Nr, @spec5 AS Spec UNION
            SELECT 6 AS Nr, @spec6 AS Spec ORDER BY 1
    OPEN c_optiuni
    FETCH NEXT FROM c_optiuni INTO @nr, @spec
    SET @gata = 0
    WHILE @@FETCH_STATUS = 0 AND @gata=0 BEGIN
            IF @spec IS NULL SET @gata=1
            ELSE
            BEGIN
                    DECLARE c_mastere CURSOR FOR
                            SELECT * FROM mastere WHERE Spec = @spec
                                    AND NrLocuri > NrLocuri_Ocupate
                            FOR UPDATE

                    OPEN c_mastere ;
                    FETCH NEXT FROM c_mastere INTO @spec_m, @denspec,
                            @nrlocuri, @nrlocuri_ocupate
                    IF @@FETCH_STATUS = 0
                            BEGIN
                            UPDATE mastere SET NrLocuri_Ocupate =
                                    NrLocuri_Ocupate + 1
                                    WHERE CURRENT OF c_mastere;
                            UPDATE candidati SET Spec_Repartizat = @spec
                                    WHERE CURRENT OF c_candidati
                            CLOSE c_mastere
                            DEALLOCATE c_mastere
                            SET @gata = 1
                            END
                    ELSE
                            BEGIN
                            CLOSE c_mastere
                            DEALLOCATE c_mastere
                            END
            END
```

```
                FETCH NEXT FROM c_optiuni INTO @nr, @spec
            END
            CLOSE c_optiuni
            DEALLOCATE c_optiuni
            FETCH NEXT FROM c_candidati INTO @idcandidat, @spec1, @spec2, @spec3,
                    @spec4, @spec5, @spec6
        END
        CLOSE c_candidati
        DEALLOCATE c_candidati
    END
```

16.5.4. Proceduri stocate în SQL PL

Nici sintaxa procedurilor redactate în IBM SQL Programming Language nu ridică probleme deosebite, odată ce am trecut prin experiența funcțiilor. Iată în listingul 16.62 procedura pentru actualizarea observațiilor pentru facturile care nu au nicio linie (în LINIIFACT).

Listingul 16.62. Procedura SQL PL ***p_update_facturi_fara_linii***

```
CREATE PROCEDURE p_update_facturi_fara_linii ()
P1: BEGIN
    UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii!'
    WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
        AND COALESCE(obs,' ') NOT LIKE '%Nu are linii!%';
END P1
```

Deosebirea cea mai importantă este, probabil, cea legată de apelul procedurilor care se realizează cu ajutorul comenzii CALL:

CALL p_update_facturi_fara_linii

Cea mai simplă rămâne procedura de ştergere în cascadă a unei facturi (listingul 16.63).

Listingul 16.63. Procedura SQL PL ***p_on_delete_cascade_factura***

```
CREATE PROCEDURE p_on_delete_cascade_factura (nrfact_ INTEGER)
    DYNAMIC RESULT SETS 1
P1: BEGIN
    DELETE FROM incasfact WHERE nrfact = nrfact_;
    DELETE FROM liniifact WHERE nrfact = nrfact_;
    DELETE FROM facturi WHERE nrfact = nrfact_;
END P1
```

La funcția ce returnează seturi de înregistrări din paragraful 16.3.4 (listingul 16.43) adăugăm procedura *f_facturi_filtrate2* din listingul 16.64[5]. Procedura declară un cursor creat pe baza unei interogări în care se face filtrarea după valoarea parametrilor de intrare și lasă acest cursor deschis pentru aplicația-client. Remarcați și clauza WITH RETURN folosită la declararea cursorului.

Listingul 16.64. Procedură SQL PL ce „returnează" un cursor-set de înregistrări

```
CREATE PROCEDURE f_facturi_filtrate2 ( codcl_ SMALLINT, data_initiala DATE,
        data_finala DATE)
```

5. Idee preluată din [IBM 2007], p.100 și [Janmohamed *et al.* 2004].

```
    DYNAMIC RESULT SETS 1
P1: BEGIN
    -- declararea cursorului
    DECLARE cursor1 CURSOR WITH RETURN FOR
        SELECT NrFact, DataFact, CodCl, Obs, ValTotala, TVA
        FROM facturi
        WHERE CodCl = COALESCE(codcl_, CodCl)
                        AND DataFact BETWEEN COALESCE(data_initiala, '2005-01-01')
                        AND COALESCE(data_finala, '2010-12-31');
    -- cursorul ramane deschis pentru aplicatia-client
    OPEN cursor1;
END P1
```

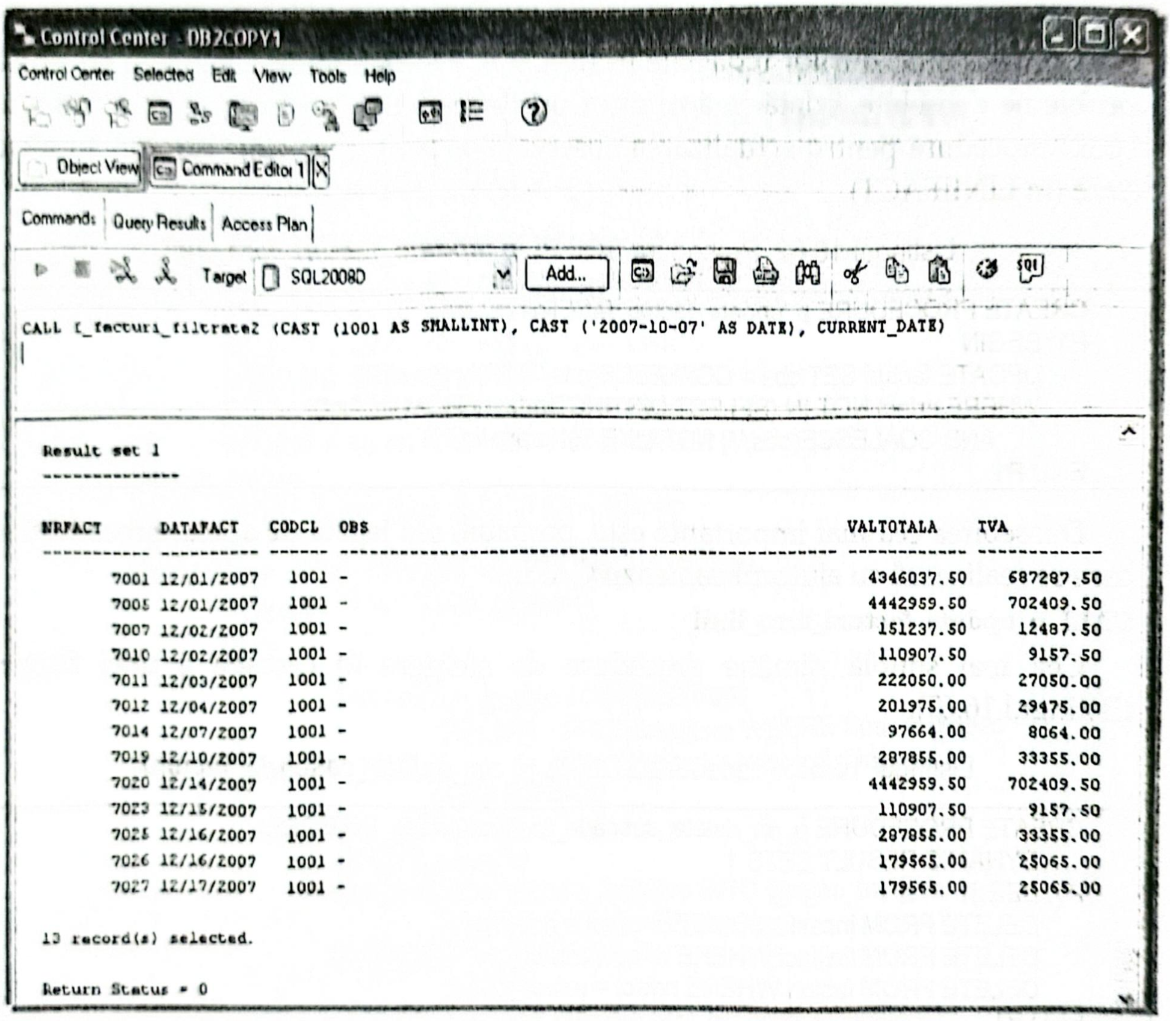


Figura 16.25. Apelul procedurii ***f_facturi_filtrate2***

Procedura nu poate fi apelată din interogări, ci doar din alte blocuri/module interne (SQL PL) sau externe (scrise în alte limbaje). Cu toate acestea, putem testa corectitudinea procedurii prin comanda CALL lansate în *DB2 Control Center* (vezi figura 16.25):

CALL f_facturi_filtrate2 (CAST (1001 AS SMALLINT),

CAST ('2007-10-07' AS DATE), CURRENT_DATE)

Încheiem, cum e şi normal, cu procedura de admitere. Crearea şi popularea celor două tabele – MASTERE şi CANDIDAŢI – se realizează în DB2 cu acelaşi

script din SQL Server (listingul 16.60). Listingul 16.65 conţine corpul procedurii p_admitere realizate în sintaxă SQL PL.

Listingul 16.65. Procedură SQL PL de admitere la masterat

```
CREATE PROCEDURE p_admitere ()
P1: BEGIN
    DECLARE v_idcandidat DECIMAL(6);
    DECLARE v_spec1 VARCHAR(4);
    DECLARE v_spec2 VARCHAR(4);
    DECLARE v_spec3 VARCHAR(4);
    DECLARE v_spec4 VARCHAR(4);
    DECLARE v_spec5 VARCHAR(4);
    DECLARE v_spec6 VARCHAR(4);
    DECLARE v_spec_repartizat VARCHAR(4);
    DECLARE v_spec VARCHAR(4);
    DECLARE v_spec_x VARCHAR(4);
    DECLARE v_nlo DECIMAL(3);
    DECLARE SQLSTATE CHAR(5);
    DECLARE v_gata SMALLINT ;

    DECLARE c_candidati CURSOR FOR
        SELECT idcandidat, spec1, spec2, spec3, spec4, spec5, spec6, spec_repartizat
        FROM candidati ORDER BY (Media_AS * 0.6 + Media_Lic * 0.4) DESC
            FOR UPDATE OF spec_repartizat;

    DECLARE c_mastere CURSOR FOR
        SELECT spec, NrLocuri_Ocupate
        FROM mastere WHERE spec = v_spec_x AND NrLocuri > NrLocuri_Ocupate
            FOR UPDATE OF NrLocuri_Ocupate;

    UPDATE mastere SET NrLocuri_Ocupate = 0;
    UPDATE candidati SET spec_repartizat = NULL;

    OPEN c_candidati;
    FETCH FROM c_candidati INTO v_idcandidat, v_spec1, v_spec2, v_spec3, v_spec4,
        v_spec5, v_spec6, v_spec_repartizat ;
    WHILE (SQLSTATE = '00000') DO
        FOR rec_optiuni AS
            SELECT v_IdCandidat, 1 AS OptiuneNr, v_Spec1 AS Spec FROM sysibm.dual UNION
            SELECT v_IdCandidat, 2 AS OptiuneNr, v_Spec2 AS Spec FROM sysibm.dual
                    WHERE v_spec2 IS NOT NULL UNION
            SELECT v_IdCandidat, 3 AS OptiuneNr, v_Spec3 AS Spec FROM sysibm.dual
                    WHERE v_spec3 IS NOT NULL UNION
            SELECT v_IdCandidat, 4 AS OptiuneNr, v_Spec4 AS Spec FROM sysibm.dual
                    WHERE v_spec4 IS NOT NULL UNION
            SELECT v_IdCandidat, 5 AS OptiuneNr, v_Spec5 AS Spec FROM sysibm.dual
                    WHERE v_spec5 IS NOT NULL UNION
            SELECT v_IdCandidat, 6 AS OptiuneNr, v_Spec6 AS Spec FROM sysibm.dual
                    WHERE v_spec6 IS NOT NULL ORDER BY 2 DO

            SET v_spec_x = rec_optiuni.spec;

            SET v_gata = 0;
            OPEN c_mastere;
            FETCH FROM c_mastere INTO v_spec, v_nlo;
            WHILE (SQLSTATE = '00000' ) AND v_gata = 0 DO
                    UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                            WHERE CURRENT OF c_mastere;
```

```
                    UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                              WHERE CURRENT OF c_candidati;
                    SET v_gata = 1;
            END WHILE ;
            CLOSE c_mastere;
    END FOR;
        FETCH FROM c_candidati INTO v_idcandidat, v_spec1, v_spec2, v_spec3, v_spec4,
            v_spec5, v_spec6, v_spec_repartizat ;
    END WHILE;
END P1
```

16.6. Pachete

Există mai multe accepţiuni ale pachetelor. Noi ne vom concentra asupra elementelor procedurale aşa cum se găsesc în Oracle PL/SQL. Fiind oarecum specifice, nici la acest subiect nu vom insista cu prea multe detalii. Folosirea pachetelor în PL/SQL are cel puţin două avantaje. Mai întâi, permite gruparea tuturor procedurilor şi funcţiilor destinate unui grup de probleme, precum un director pe disc în care se grupează o serie de fişiere. În al doilea rând, toate variabilele declarate în specificaţiile unui pachet sunt publice.

Începem cu un pachet care are aceeaşi funcţionalitate cu funcţia *f_facturi_filtrate_1* din paragraful 16.3.2 (listingul 16.37) ce returnează un set de înregistrări. Orice pachet are două părţi. Prima - *specificaţiile* - este publică şi obligatorie, conţinând variabile, cursoare şi excepţii, precum şi elementele de identificare (nume, parametri şi tipuri returnate, în cazul funcţiilor) ale unor eventuale proceduri şi/sau funcţii. A doua - *corpul* - este „privată" (ascunsă), fiind cea în care este descris corpul procedurilor şi funcţiilor declarate în specificaţii. Cele două părţi ale pachetului *pac_seturi* din listingul 16.66 nu aduc neapărat un plus de productivitate, ci constituie doar o alternativă la varianta din paragraful 16.3.2.

Listingul 16.66. Primul pachet PL/SQL

```
---------- specificatiile (antetul) pachetului
CREATE OR REPLACE PACKAGE pac_seturi AS
 TYPE t_ref_facturi IS REF CURSOR;
 FUNCTION f_facturi_filtrate1
        (data_initiala DATE, data_finala DATE)
        RETURN t_ref_facturi;
END;
/
---------- corpul pachetului
CREATE OR REPLACE PACKAGE BODY pac_seturi AS
--------------------------------------------------------
 FUNCTION f_facturi_filtrate1 (data_initiala DATE,
  data_finala DATE) RETURN t_ref_facturi
AS
 set_of_facturi t_ref_facturi;
BEGIN
 OPEN set_of_facturi FOR SELECT * FROM facturi WHERE datafact
        BETWEEN data_initiala AND data_finala;
 RETURN set_of_facturi ;
```

```
END f_facturi_filtrate1;

END; -- aici se termina corpul pachetului
/
```

Tipul *t_ref_facturi* este declarat în specificații, fiind public. Devine astfel posibilă folosirea sa la definirea (în clauza RETURN) funcției *f_facturi_filtrate1*. Apelul funcției aflate în pachet presupune prefixarea numelui ei cu numele pachetului:

SELECT pac_seturi.f_facturi_filtrate1(DATE'2007-08-15', CURRENT_DATE)
FROM dual

În continuare, funcții şi proceduri descrise în paragrafele anterioare – *f_proctva, f_valtot, f_val_cumul, p_cumul, p_cumul2* - sunt incluse într-un pachetul *pac_vanzari* (vezi listingul 16.67). Funcția *f_proctva* are două semnături. Dacă la invocarea sa, parametrul trimis este numeric, PL/SQL „înțelege" că acesta este un cod de produs şi se execută prima „instanță a" funcției, cea care returnează procentul de TVA al produsului cu *codul* specificat. Dacă, la invocare, parametrul este de tip şir de caractere, atunci se va executa a doua instanță a funcției, cea care returnează procentul de TVA al unui produs cu *denumirea* specificată. Fireşte, această a doua instanță a funcției funcționează corect doar dacă nu există în tabela PRODUSE cel puțin două sortimente cu denumire identică. În caz contrar, SELECT-ul va identifica mai multe linii în tabelă, iar funcția va genera o eroare.

Listingul 16.67. Antetul pachetului ***pac_vanzari***

```
-- in pachetul PAC_VANZARI declaram patru variabile publice
CREATE OR REPLACE PACKAGE pac_vanzari AS
-- acestea sunt cele patru variabile publice
  v_trg_liniifact BOOLEAN := FALSE;
  v_trg_incasfact BOOLEAN := FALSE;
  v_nrfact liniifact.nrfact%TYPE;
  v_linie liniifact.linie%TYPE;

    FUNCTION f_proctva (codpr_ produse.codpr%TYPE) RETURN produse.proctva%TYPE ;
    FUNCTION f_proctva (denpr_ produse.denpr%TYPE) RETURN produse.proctva%TYPE ;
    FUNCTION f_valtot (nrfact_ facturi.nrfact%TYPE)    RETURN NUMERIC;
    FUNCTION f_val_cumul (nrfact_ facturi.nrfact%TYPE) RETURN NUMERIC;
    PROCEDURE p_cumul;
    PROCEDURE p_cumul2;
END; -- pac_vanzari
/
----------------------------------------------------------------
CREATE OR REPLACE PACKAGE BODY pac_vanzari AS
----------------------------------------------------------------
-- functie ce returneaza procentul TVA pentru un cod produs dat
FUNCTION f_proctva (codpr_ produse.codpr%TYPE)
        RETURN produse.proctva%TYPE
IS
    v_proc produse.proctva%TYPE;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_;
    RETURN v_proc;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
```

```
    WHEN OTHERS THEN
        RETURN NULL;
END f_proctva;

------------------------------------------------------------------------
-- aceeasi functie ce returneaza procentul TVA pentru o denumire de produs data
FUNCTION f_proctva (denpr_ produse.denpr%TYPE)
        RETURN produse.proctva%TYPE
IS
    v_proc produse.proctva%TYPE;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE denpr = denpr_;
    RETURN v_proc;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL;
END f_proctva;

------------------------------------------
-- functie ce returneaza valoarea cu TVA a unei facturi
FUNCTION f_valtot (nrfact_ facturi.nrfact%TYPE)
        RETURN NUMERIC
IS
    v_valfact NUMERIC(14,2);
BEGIN
    SELECT SUM(cantitate * pretunit * (1 + f_proctva(codpr)))
        INTO v_valfact FROM liniifact WHERE nrfact = nrfact_;
    RETURN v_valfact;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL;
END f_valtot;

------------------------------------------
-- functie ce returneaza suma valorilor facturilor precedente (unei facturi)
FUNCTION f_val_cumul (nrfact_ facturi.nrfact%TYPE)
    RETURN NUMERIC
IS
    v_valtot NUMERIC(16,2);
BEGIN
    SELECT SUM(cantitate * pretunit * (1 + f_proctva(codpr)))
        INTO v_valtot
    FROM liniifact
    WHERE nrfact <= nrfact_;
    RETURN v_valtot;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL;
END f_val_cumul;

------------------------------------------
-- procedura de afisare a valorii facturii curente si sumei valorilor facturilor precedente
-- varianta 2 - cursor
```

```
PROCEDURE p_cumul
IS
    CURSOR c_fact IS
        SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
        FROM facturi;
    rec_fact c_fact%ROWTYPE;
    v_suma NUMBER(16,2) := 0;
BEGIN
    OPEN c_fact;
    FETCH c_fact INTO rec_fact;
    LOOP
        EXIT WHEN c_fact%NOTFOUND;
        v_suma := v_suma + NVL(rec_fact.valfact,0);
        DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                TO_CHAR(v_suma, '999999999999.99') );
        FETCH c_fact INTO rec_fact;
    END LOOP;
    CLOSE c_fact;
END p_cumul;

------------------------------------------------
-- procedura de afisare a valorii facturii curente si sumei valorilor facturilor precedente
-- varianta 3 - (TOT) cursor
PROCEDURE p_cumul2
IS
    v_suma NUMBER(16,2) := 0;
BEGIN
    FOR rec_fact IN (
                SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
                FROM facturi ) LOOP
        v_suma := v_suma + NVL(rec_fact.valfact,0);
        DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                TO_CHAR(v_suma, '999999999999.99') );
    END LOOP;
END p_cumul2;

END; -- sfirsit corp pachet pac_vanzari
```

Chiar dacă ne putem declara mulțumiți pentru că procedurile și funcțiile sunt mai adunate, plus că avem la dispoziție câteva variabile publice pe care le vom pune la lucru în capitolul următor, trebuie să observăm și un dezavantaj al unor pachete. Atunci când lungimea lor crește, sunt mai greu de urmărit și orice eroare strecurată într-o funcție sau procedură atrage după sine invalidarea întregului pachet.

Capitolul 17
Declanşatoare

Declanşatoarele (*trigger*-e, în original) reprezintă o categorie deosebită de proceduri stocate ale unei baze de date. Particularitatea lor esenţială ţine de faptul că, odată create şi stocate în schema bazei, acestea sunt executate *automat* la operaţiuni pentru care au fost definite. Tradiţional, sunt trei tipuri de astfel de operaţiuni: inserare, modificare şi ştergere ale liniilor din tabela pentru care au fost definite. La această tipologie a declanşatoarelor, unele SGBD-uri au mai adăugat şi alte categorii, asociate, spre exemplu, actualizărilor tabelelor virtuale, conectării unei aplicaţii/utilizator la baza de date, deschiderii sau închiderii bazei de date etc.

Pentru lucrarea de faţă, problema principală a declanşatoarelor ţine de faptul că acestea sunt scrise în extensiile procedurale ale SQL care, ca şi funcţiile şi procedurile din capitolul anterior, prezintă diferenţe sensibile de la un produs la altul: PL/SQL (Oracle), SQL PL (DB2), Transact-SQL (SQL Server), PL/pgSQL etc.

Cele mai importante avantaje ale declanşatoarelor sunt:

- permit instituirea unor reguli de validare cu mult mai complexe decât posibilităţile clauzei CHECK;
- pot ameliora sensibil mecanismul de securitate al bazei;
- permit instituirea (acolo unde nu este posibil prin clauza FOREIGN KEY) restricţiilor referenţiale şi a modului de tratare a modificărilor ce pot cauza probleme de integritate referenţială;
- constituie suportul calculării automate a valorilor unor atribute, ceea ce poate atrage un spor de viteză considerabil;
- posibilitatea jurnalizării explicite, în formatul dorit a tuturor operaţiunilor de tip DML şi DML dintr-o (sub)schemă a bazei de date;
- sincronizarea replicilor unei baze de date distribuite/replicate.

17.1. Tipologia declanşatoarelor

Aşadar, aria de folosire a declanşatoarelor este foarte extinsă. Totuşi, documentaţiile de produs recomandă ca elanul *trigger*-esc să nu depăşească anumite limite. Prin declanşatoare trebuie gestionate numai acele reguli imposibil de asigurat prin restricţiile la nivel de atribut (câmp), înregistrare, restricţii de tip PRIMARY KEY, NOT NULL etc. Tipologia „clasică" numără nu mai puţin de 15 categorii de triggere (vezi tabelul 17.1).

Tabelul 17.1. Tipologia „clasică" a declanşatoarelor

Eveniment declanşator	Declanşare pentru fiecare	Descriere
BEFORE INSERT	Comandă	Codul (programul) acestuia se lansează înaintea executării unei comenzi INSERT în tabela-ţintă
BEFORE INSERT	Linie	Se execută înaintea inserării fiecărei linii în tabelă
AFTER INSERT	Linie	Se execută după inserarea fiecărei linii în tabelă
AFTER INSERT	Comandă	Se lansează după execuţia unei comenzi de inserare de linii în tabelă
INSTEAD OF INSERT	Linie	În loc să se insereze o linie în tabelă, se execută codul din acest declanşator
BEFORE UPDATE	Comandă	Codul acestuia se execută înaintea executării unei comenzi UPDATE pentru tabela-ţintă
BEFORE UPDATE	Linie	Se execută înaintea modificării fiecărei linii din tabelă
AFTER UPDATE	Linie	Se execută după modificarea fiecărei linii
AFTER UPDATE	Comandă	Se lansează după execuţia comenzii UPDATE (după modificarea tuturor liniilor afectate de comandă)
INSTEAD OF UPDATE	Linie	În loc să se modifice o linie din tabela-ţintă, se execută codul din acest declanşator
BEFORE DELETE	Comandă	Se execută înaintea comenzii DELETE
BEFORE DELETE	Linie	Se execută înainte de ştergerea fiecărei linii
AFTER DELETE	Linie	Se execută după ştergerea fiecărei linii
AFTER DELETE	Comandă	Se lansează după execuţia comenzii DELETE (după ştergerea tuturor liniilor afectate de comandă)
INSTEAD OF DELETE	Linie	Se execută în locul ştergerii unei linii din tabela-ţintă

Declanşatoare la nivel de linie şi la nivel de comandă (statement)

La definirea *trigger*-ului, se poate stabili de câte ori este executat acesta: de fiecare dată când o linie este inserată/modificată/ştearsă sau o dată pentru o comandă de actualizare, indiferent de câte linii sunt afectate. Să luăm, spre exemplu, comanda:

UPDATE clienti SET codcl = codcl + 100 WHERE codcl > 3

Presupunând că 135 de clienţi au codul mai mare decât 3, un *trigger* de modificare la nivel de linie se execută de 135 de ori, în timp ce unul la nivel de comandă o singură dată. Dacă nici un client nu are codul mai mare de 3, un *trigger* de modificare la nivel de linie nu s-ar putea executa deloc, în timp ce declanşatorul declarat a fi la nivel de comandă ar fi lansat (evident, o singură dată).

Declanşatoare de tip înainte (BEFORE) şi după (AFTER) actualizare

Un *trigger* de tip BEFORE este ideal pentru verificarea unui set de condiţii care ar putea bloca de la bun început tranzacţia respectivă (autentificări, soldul 0 pentru un cont sau pentru un material într-o magazie etc.). Dacă pentru tabelă au fost definite şi restricţii (NOT NULL, PRIMARY KEY, CHECK), dar şi declanşatoare de tip BEFORE, declanşatoarele BEFORE se lansează *înaintea* verificării restricţiilor. Declanşatoarele de tip AFTER se execută după verificarea restricţiilor, fiind nimerite la jurnalizare (ce ţine evidenţa strictă a modificărilor operate în tabele cu informaţii sensibile – spre exemplu, calculul automat al costului mediu ponderat al unui material după fiecare intrare în magazie sau calculul soldului curent al unui cont bancar după fiecare depunere sau retragere etc.). Spre deosebire de cele de tip BEFORE, *trigger*-ele de tip AFTER blochează liniile procesate. Dacă declanşatoarele prelucrează/interoghează tabelele pe care au fost denumite[1], declanşatoarele de tip BEFORE nu „văd" modificările care atrag declanşarea lor, în timp ce *trigger*-ele AFTER „văd" imaginea bazei de date după modificare.

Orice declanşator la nivel de linie sau comandă poate fi de tip BEFORE sau AFTER. Astfel, pentru orice tabelă şi operaţie de inserare/modificare/ştergere pot fi definite patru tipuri de *trigger*-e care se execută în ordinea:

- înainte – la nivel de comandă (*BEFORE-statement*);
- înainte – la nivel de linie (*BEFORE-row*);
- după – la nivel de linie (*AFTER-row*);
- după – la nivel de comandă (*AFTER-statement*).

Trigger-*e de tip în-loc-de (INSTEAD OF)*

Acest gen de declanşator constituie un excelent mijloc de propagare a modificărilor dintr-o tabelă virtuală în tabelele de bază din care provine. Amânăm această discuţie pentru finalul fiecărui paragraf, începând cu următorul.

Un alt element important este că şi în *trigger*-ele la nivel de linie (ROW) de tip BEFORE, şi în cele de tip AFTER, pot fi invocate şi prelucrate atât valorile atributelor dinaintea operaţiei, cât şi cele de după operaţie. Prefixul este, după caz, :OLD/OLD sau :NEW/NEW.

17.2. Declanşatoare în PL/pgSQL

Limbajul PL/pgSQL este generos în materie de declanşatoare. Să începem cu unul simplu dedicat cheilor surogat. Atributul CodCl din tabela CLIENTI este o asemenea cheie surogat, întrucât valorile sale nu au nicio legătură cu valorile celorlalte atribute ale fiecărui client, fiind mai mult un soi de număr curent. Primii clienţi au fost inseraţi încă din capitolul 3. Din acest moment, dorim ca, la inserare, codul fiecărui client să nu mai depindă de utilizator (valoarea specificată în

1. Nu întotdeauna este permisă interogarea unei tabele pentru care s-au definit declanşatoare la nivel de linie (ROW), mai ales în caz declanşatoarelor de tip UPDATE. Cei dintre dumneavoastră care au lucrat în Oracle, au avut, probabil, de a face cu problema *mutanţei* tabelei.

comanda INSERT), ci să fie atribuită automat de sistem. Pentru aceasta, vom crea un obiect al bazei de date de care am pomenit în capitolul 2, şi anume o secvenţă:

CREATE SEQUENCE seq_codcl START WITH 1011 INCREMENT BY 1

MINVALUE 1001 MAXVALUE 8900 NO CYCLE;

O secvenţă este un obiect dintr-o bază de date care generează o valoare unică atunci când i se cere. Chiar şi atunci când câteva mii de utilizatori ar introduce simultan clienţi, vrem să fim siguri că valoarea fiecăruia este unică. Or secvenţa este construită astfel ca eventualele cereri de valori unice să fie serializate. Definim acum un declanşator de tip BEFORE-INSERT-ROW (vezi listingul 17.1). PL/pgSQL reclamă crearea unei funcţii care să returneze TRIGGER şi apoi o comandă CREATE TRIGGER care să folosească funcţia respectivă drept argument.

Listingul 17.1. Funcţia-declanşator PL/pgSQL executată la inserarea unei înregistrări în tabela CLIENŢI

```
CREATE OR REPLACE FUNCTION trg_clienti_ins_before_row()
    RETURNS TRIGGER AS $trg_clienti_ins_before_row$
BEGIN
    NEW.CodCl = NEXTVAL('seq_codcl');
    RETURN NEW;
END;
$trg_clienti_ins_before_row$ LANGUAGE plpgsql;
```

Extragerea următoarei valori unice disponibile din secvenţă se realizează prin funcţia *NEXTVAL*. *NEW.CodCl* semnalizează că valoarea generată de secvenţă va fi atribuită atributului *CodCl* de pe linia nou-introdusă. Funcţia se termină obligatoriu prin *RETURN NEW*, ceea ce înseamnă că se returnează noua linie inserată în care au fost făcute eventuale modificări. Asociem funcţia declanşatorului printr-o comandă CREATE TRIGGER:

CREATE TRIGGER trg_clienti_ins_before_row

BEFORE INSERT ON clienti

FOR EACH ROW EXECUTE PROCEDURE trg_clienti_ins_before_row()

Reţinem că modificarea valorii unuia sau mai multor atribute de pe linia proaspăt inserată este posibilă doar în declanşatoare *BEFORE-ROW*. Acum să testăm funcţionalitatea declanşatorului. Introducem un client:

INSERT INTO clienti VALUES (999, 'Client destul de nou', 'R12345',

'Adresa clientului nou', 700505, NULL)

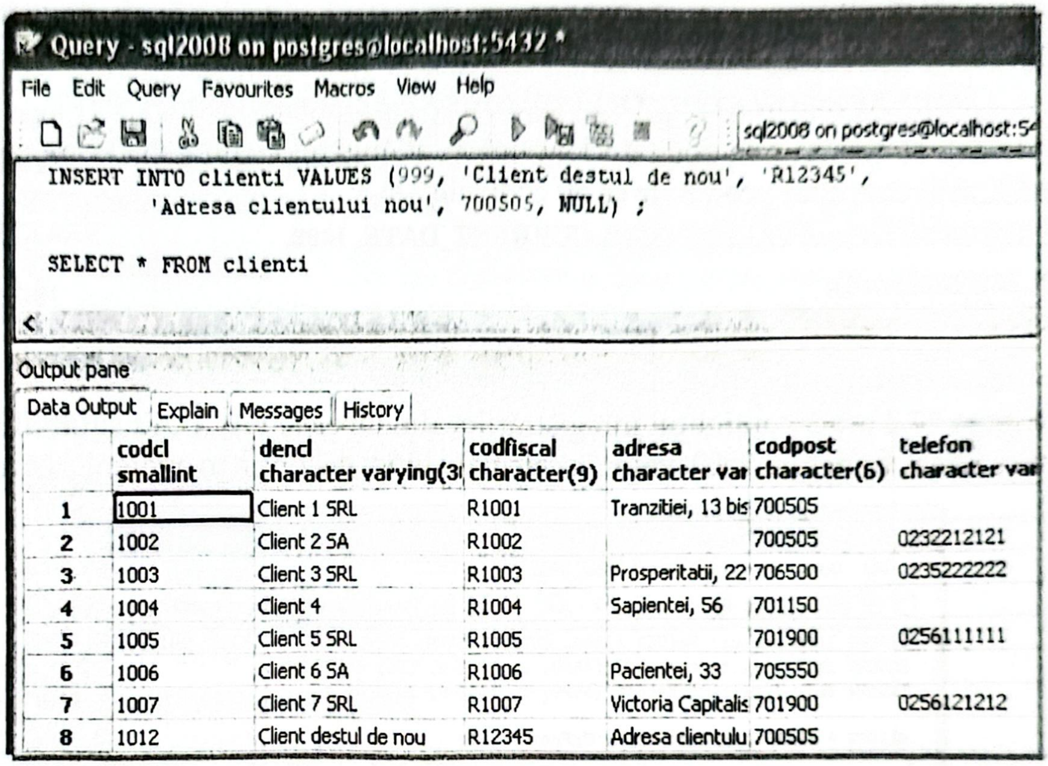
Valoarea *CodCl* specificată în clauza VALUES este 999. Dacă însă vizualizăm noua înregistrare (ultima din figura 17.1) observăm că PostgreSQL a ţinut cont de declanşator, şi nu de ceea ce am specificat noi în comanda INSERT.

Continuăm cu declanşatorul de inserare în tabela FACTURI. Pentru a fi siguri că orice factură nou-introdusă porneşte de la zero cu valorile totală, încasată şi TVA, declarăm un declanşator după cum urmează (listingul 17.2).

Listingul 17.2. Declanşatorul PL/pgSQL pentru inserări în tabela FACTURI

```
CREATE OR REPLACE FUNCTION trg_facturi_ins()
    RETURNS TRIGGER AS $trg_facturi_ins$
BEGIN
```

```
        NEW.valtotala := 0;
        NEW.tva := 0;
        NEW.valincasata := 0;

        RETURN NEW;
    END;
    $trg_facturi_ins$ LANGUAGE plpgsql;
```



	codcl smallint	dencl character varying(3	codfiscal character(9)	adresa character var	codpost character(6)	telefon character var
1	1001	Client 1 SRL	R1001	Tranzitiei, 13 bis	700505	
2	1002	Client 2 SA	R1002		700505	0232212121
3	1003	Client 3 SRL	R1003	Prosperitatii, 22	706500	0235222222
4	1004	Client 4	R1004	Sapientei, 56	701150	
5	1005	Client 5 SRL	R1005		701900	0256111111
6	1006	Client 6 SA	R1006	Pacientei, 33	705550	
7	1007	Client 7 SRL	R1007	Victoria Capitalis	701900	0256121212
8	1012	Client destul de nou	R12345	Adresa clientulu	700505	

Figura 17.1. Verificarea declanşatorului de inserare în CLIENȚI

CREATE TRIGGER trg_facturi_ins BEFORE INSERT ON facturi
FOR EACH ROW EXECUTE PROCEDURE trg_facturi_ins();

Atributele *ValTotală* şi *TVA* ale tabelei FACTURI trebuie recalculate după fiecare inserare (eventual) modificare şi ştergere a unei linii din factura respectivă, altfel spus la INSERT-uri, DELETE-uri şi unele UPDATE-uri asupra tabelei LINIIFACT. Aşa încât, ca model, vom redacta declanşatorul de inserare în tabela LINIIFACT (vezi listingul 17.3), care modifică valorile atributelor:

- *Linie* şi *TVALinie* din LINIIFACT;
- *TVA* şi *VALTotală* din FACTURI.

Listingul 17.3. Declanşatorul PL/pgSQL de inserare în tabela LINIIFACT

```
DROP FUNCTION trg_liniifact_ins() CASCADE;

CREATE OR REPLACE FUNCTION trg_liniifact_ins()
    RETURNS TRIGGER AS $trg_liniifact_ins$
BEGIN
    NEW.tvalinie := COALESCE(NEW.cantitate,0) * COALESCE(NEW.pretunit,0) *
        f_proctva(NEW.codpr);
    NEW.linie := (SELECT COALESCE(MAX(Linie),0) + 1 FROM liniifact
        WHERE NrFact = NEW.NrFact);
```

```
        UPDATE facturi SET
            tva = COALESCE(tva,0) + NEW.tvalinie,
            valtotala = COALESCE(valtotala,0) + NEW.cantitate * NEW.pretunit + NEW.tvalinie
            WHERE nrfact = NEW.nrfact;
        RETURN NEW;
    END;
    $trg_liniifact_ins$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_liniifact_ins BEFORE INSERT ON liniifact

FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_ins();

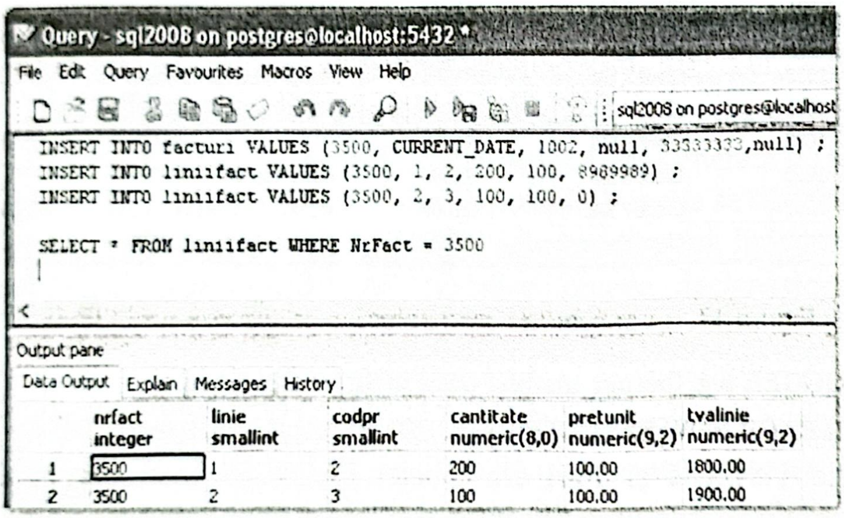
Haideţi să testăm cele două declanşatoare, adăugând factura 3500 care conţine două linii în care apar produsele ce au codurile 2 şi 3:

INSERT INTO facturi VALUES (3500, CURRENT_DATE, 1002, null, 33333333,null);

INSERT INTO liniifact VALUES (3500, 1, 2, 200, 100, 8989989);

INSERT INTO liniifact VALUES (3500, 2, 3, 100, 100, 0);

Figura 17.2 arată conţinutul înregistrărilor din tabelele FACTURI şi LINIIFACT care se referă la factura 3500, valorile calculate fiind, cel puţin în aparenţă, corecte.



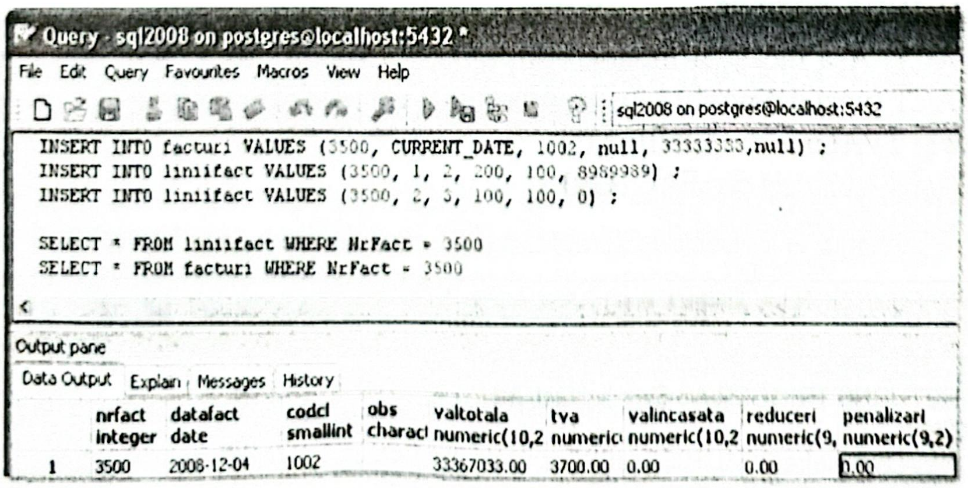


Figura 17.2. Verificarea declanşatoarelor de inserare în tabelele FACTURI şi LINIIFACT

Dacă la inserarea unei linii dintr-o factură (tabela LINIIFACT), valorile facturate şi ale TVA (din FACTURI) trebuie incrementate, la ştergerea unei linii, valorile celor două atribute trebuie scăzute. Poate apărea, însă, şi o problemă de numerotare a liniilor. Astfel, dacă factura are cinci linii, iar linia ştearsă este a doua, în urma ştergerii după linia 1 urmează 3. Acest gol ar putea fi supărător pentru unii, aşa că avem două variante. Prima este cea restrictivă: nu se permite decât ştergerea ultimei linii dintr-o factură (vezi listingul 17.4).

Listingul 17.4. Declanşatorul PL/pgSQL de ştergere în tabela LINIIFACT – versiunea 1.0

```
CREATE OR REPLACE FUNCTION trg_liniifact_del()
    RETURNS TRIGGER AS $trg_liniifact_del$
DECLARE
    v_nrlinii NUMERIC(3);
BEGIN
    SELECT MAX(linie) INTO v_nrlinii FROM liniifact WHERE nrfact = OLD.nrfact;
    IF OLD.linie <> COALESCE(v_nrlinii,0) THEN
        RAISE EXCEPTION 'Nu puteti sterge decat ultima linie dintr-o factura ';
    END IF;

    UPDATE facturi
        SET tva = tva - OLD.tvalinie,
                valtotala = valtotala - OLD.cantitate * OLD.pretunit
                - OLD.tvalinie
        WHERE nrfact = OLD.nrfact;
    RETURN OLD;
END;
$trg_liniifact_del$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_liniifact_del BEFORE DELETE ON liniifact
        FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_del();
```

Dacă după crearea acestui declanşator, încercăm să ştergem prima linie a facturii 3500:

```
DELETE FROM liniifact WHERE NrFact = 3500 AND linie = 1
```

obţinem mesajul de eroare din figura 17.3.

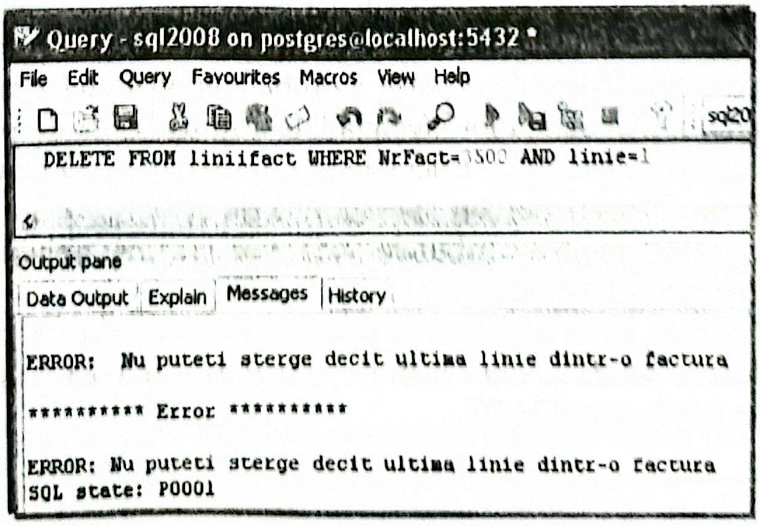


Figura 17.3. Verificarea declanşatorului de ştergere în tabela LINIIFACT (v1.0)

Ne simţim vinovaţi pentru rigiditatea soluţiei, aşa că ne vom gândi la o alta, în care utilizatorul poate şterge orice linie dintr-o factură, iar liniile vor fi renumerotate automat. Iată cum ar putea arăta această versiune (listingul 17.5).

Listingul 17.5. Declanşatorul PL/pgSQL de ştergere în tabela LINIIFACT – versiunea 2.0

```
-- varianta 2 - se permite stergerea oricarei linii si se renumeroteaza liniile acelei facturi

DROP FUNCTION trg_liniifact_del() CASCADE;

CREATE OR REPLACE FUNCTION trg_liniifact_del()
    RETURNS TRIGGER AS $trg_liniifact_del$
DECLARE
    v_liniemax liniifact.linie%TYPE;
BEGIN
    SELECT MAX(linie) INTO v_liniemax FROM liniifact
WHERE nrfact = OLD.NrFact;

    IF v_liniemax = OLD.linie THEN
        NULL;
    ELSE
        UPDATE liniifact SET linie = linie - 1 WHERE linie > OLD.linie
                AND NrFact = OLD.NrFact;
    END IF;

    UPDATE facturi
    SET tva = tva - OLD.tvalinie,
        valtotala = valtotala - OLD.cantitate * OLD.pretunit - OLD.tvalinie
    WHERE nrfact = OLD.nrfact;

    RETURN OLD;
END;
$trg_liniifact_del$ LANGUAGE plpgsql;
CREATE TRIGGER trg_liniifact_del AFTER DELETE ON liniifact
FOR EACH ROW
        EXECUTE PROCEDURE trg_liniifact_del();
```

Testăm validitatea declanşatorului, ştergând prima linie din facturra 3500[2], iar după cum o arată figura 17.4, de exemplu, a doua linie a devenit prima:

DELETE FROM liniifact WHERE NrFact = 3500 AND linie = 1

Până acum am rezolvat doar o parte a problemei atributelor calculate, cea mai spumoasă, şi anume calcularea lor efectivă. Ne-a rămas ceva de pus la punct, fără de care corectitudinea atributelor calculate ar fi îndoielnică. Să imaginăm o situaţie în care s-ar cuveni să restricţionăm operaţiunile de actualizare a tabelei FACTURI. Modificările valorilor pentru atributele *ValTotală* şi *TVA* trebuie să fie permise numai dacă sunt consecinţă a declanşatoarelor tabelei LINIIFACT. Dacă nu am face o asemenea verificare, orice utilizator ar putea să modifice direct (printr-o comandă UPDATE) valoarea totală a unei facturi şi, astfel, s-ar crea o diferenţă faţă de informaţiile din tabela LINIIFACT.

2. Cele două linii ale facturii 3550 dinainte acestei ştergeri sunt vizibile în figura 17.2.

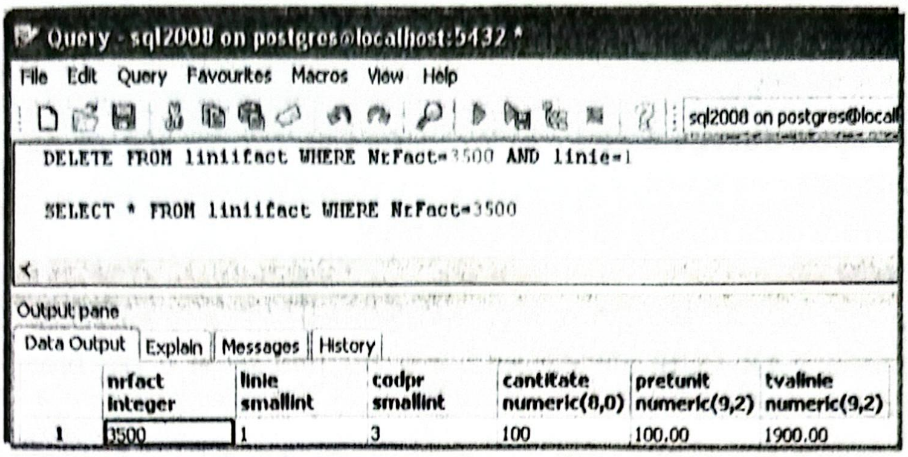


Figura 17.4. Verificarea declanşatorului de ştergere în tabela LINIIFACT (v2.0)

Ideea este una simplă: atunci când se modifică valoarea atributelor *ValTotală* şi *TVA*, declanşatorul de modificare (UPDATE) din FACTURI verifică dacă modificarea provine din unul dintre declanşatoarele (INSERT, UPDATE sau DELETE) tabelei LINIIFACT. Dacă sursa modificării este un declanşator din LINIIFACT, modificarea este autorizată; în caz contrar, este interzisă.

În PostgreSQL nu există variabile publice, aşa că vom folosi o tabelă pe care o vom numi, ostentativ, VAR_PUB:

```
CREATE TABLE var_pub (
        utilizator VARCHAR(40) NOT NULL,
        v_trg_liniifact BOOLEAN NOT NULL DEFAULT FALSE,
        v_trg_incasfact BOOLEAN NOT NULL DEFAULT FALSE
                        );
```

Tabela va avea câte o linie pentru fiecare utilizator conectat la baza de date. Celelalte două atribute, ambele de tip BOOLEAN[3], vor avea valoarea logică TRUE dacă este în curs de execuţie unul dintre declanşatoarele tabelei LINIIFACT, respectiv ÎNCASFACT. Vă reamintesc (am discutat pe acest subiect şi prin capitolul 6) că putem afla numele utilizatorului curent conectat la baza de date prin funcţia sistem USER sau CURRENT_USER:

```
SELECT USER
```

Întrucât nu putem anticipa care sunt utilizatorii ce vor lucra cu baza de date, putem gândi o procedură (ştiţi deja că în PL/pgSQL procedurile sunt tot funcţii) care să creeze, la apelare, o înregistrare în tabela VAR_PUB, dacă această înregistrare nu exista dinainte. Iată funcţia-procedură în listingul 17.6.

Listingul 17.6. Funcţia PL/pgSQL care inserează în VAR_PUB, dacă e nevoie, o înregistrare pentru utilizatorul curent

```
CREATE OR REPLACE FUNCTION p_init_vp () RETURNS VOID AS $$
BEGIN
    IF NOT EXISTS (SELECT * FROM var_pub WHERE utilizator = USER) THEN
        INSERT INTO var_pub VALUES (USER, FALSE, FALSE);
```

3. Îmi face plăcere să vă reamintesc că, dintre cele patru servere BD, doar PostgreSQL acceptă în tabele atribute de tip BOOLEAN.

```
    END IF;
END;
$$LANGUAGE plpgsql;
```

Pentru a o apela direct, folosim:

SELECT p_init_vp()

Putem verifica dacă funcţia funcţionează prin:

SELECT * FROM var_pub

În continuare (vezi listingul 17.7) creăm două funcţii ce furnizează valorile curente (din tabela VAR_PUB) ale atributelor *v_trg_liniifact* şi *v_trg_încasfact* pentru utilizatorul curent.

Listingul 17.7. Funcţii PL/pgSQL ce extrag valorile atributelor *v_trg_liniifact* şi *v_trg_încasfact* pentru utilizatorul curent

```
CREATE OR REPLACE FUNCTION f_trg_liniifact () RETURNS BOOLEAN AS $$
DECLARE
 v BOOLEAN := FALSE;
BEGIN
 SELECT v_trg_liniifact INTO v FROM var_pub WHERE utilizator = USER;
 RETURN v;
END;
$$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION f_trg_incasfact () RETURNS BOOLEAN AS $$
DECLARE
 v BOOLEAN := FALSE;
BEGIN
 SELECT v_trg_incasfact INTO v FROM var_pub WHERE utilizator = USER;
 RETURN v;
END;
$$ LANGUAGE plpgsql;
```

Tot pentru a ne uşura munca vom crea o funcţie-procedură care va „seta" valoarea unuia dintre cele două câmpuri pe TRUE sau FALSE (vezi listingul 17.8).

Listingul 17.8. Funcţie-procedură de setare a valorilor atributelor *v_trg_liniifact* şi *v_trg_încasfact* pentru utilizatorul curent

```
CREATE OR REPLACE FUNCTION p_set (atribut_ VARCHAR, valoare_ BOOLEAN)
    RETURNS VOID AS $$
BEGIN
 IF UPPER(atribut_) = 'V_TRG_LINIIFACT' THEN
    UPDATE var_pub SET v_trg_liniifact = valoare_ WHERE utilizator = USER;
 ELSE
    IF UPPER(atribut_) = 'V_TRG_INCASFACT' THEN
        UPDATE var_pub SET v_trg_incasfact = valoare_ WHERE utilizator = USER;
    END IF;
 END IF;
END;
$$ LANGUAGE plpgsql;
```

Toate aceste funcţii constituie accesorii pentru seria declanşatoarelor care urmează. Începem cu declanşatorul de inserare în LINIIFACT (listingul 17.9). Păstrăm varianta anterioară, la care adăugam un apel (prin comanda PERFORM) la

procedura *p_init_vp*, şi setarea pe TRUE a valorii atributului *v_trg_liniifact* pentru utilizatorul curent. După incrementarea valorilor atributelor *ValTotală* şi *TVA*, readucem valoarea atributului *v_trg_liniifact* pentru utilizatorul curent la „starea" FALSE.

Listingul 17.9. O nouă versiune a declanşatorului de inserare a unei înregistrări în LINIIFACT

```
DROP FUNCTION trg_liniifact_ins() CASCADE;
CREATE OR REPLACE FUNCTION trg_liniifact_ins() RETURNS TRIGGER AS $trg_liniifact_ins$
BEGIN
    NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);

    -- noutate !!!
    PERFORM p_init_vp();
    PERFORM p_set ('v_trg_liniifact', TRUE);
    UPDATE facturi SET tva = tva + NEW.tvalinie,
                valtotala = valtotala + NEW.cantitate * NEW.pretunit + NEW.tvalinie
        WHERE nrfact = NEW.nrfact;

    -- refacem valoarea pseudo-variabilei
    PERFORM p_set ('v_trg_liniifact', FALSE);
    RETURN NEW;
END;
$trg_liniifact_ins$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_liniifact_ins BEFORE INSERT ON liniifact

FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_ins();

Declanşatorul de modificare a unei linii în LINIIFACT este ceva mai interesant, pentru că e prima versiune discutată (în listingul 17.10).

Listingul 17.10. Noua (de fapt, e prima!) versiune a declanşatorului de modificare a unei înregistrări din LINIIFACT

```
DROP FUNCTION trg_liniifact_upd_br() CASCADE;

CREATE OR REPLACE FUNCTION trg_liniifact_upd_br()
    RETURNS TRIGGER AS $trg_liniifact_upd_br$
BEGIN
    IF NEW.nrfact = OLD.nrfact THEN           -- nu s-a schimbat nrfact !!!!
        IF NEW.codpr <> OLD.codpr OR NEW.cantitate <> OLD.cantitate OR
                  NEW.pretunit <> OLD.pretunit OR NEW.tvalinie <> OLD.tvalinie THEN
            NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);

            PERFORM p_init_vp();
            PERFORM p_set ('v_trg_liniifact', TRUE);

            UPDATE facturi SET tva = tva - OLD.tvalinie + NEW.tvalinie,
                    valtotala = valtotala - (OLD.cantitate * OLD.pretunit + OLD.tvalinie)
                                + (NEW.cantitate * NEW.pretunit + NEW.tvalinie)
            WHERE nrfact = NEW.nrfact;

            PERFORM p_set ('v_trg_liniifact', FALSE);

        END IF;
    ELSE -- vechile valori trebuie scazute de la "vechea" factura
        -- iar noile valori adaugate "noii" facturi
        PERFORM p_init_vp();
```

```
            PERFORM p_set ('v_trg_liniifact', TRUE);
            UPDATE facturi SET tva = tva - OLD.tvalinie,
                           valtotala = valtotala - (OLD.cantitate * OLD.pretunit + OLD.tvalinie)
            WHERE nrfact = OLD.nrfact;

            NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);
            UPDATE facturi SET tva = tva + NEW.tvalinie,
                           valtotala = valtotala + (NEW.cantitate * NEW.pretunit + NEW.tvalinie)
            WHERE nrfact = NEW.nrfact;

            -- refacem valoarea pseudo-variabilei
            PERFORM p_set ('v_trg_liniifact', FALSE);
        END IF;
        RETURN NEW;
    END;
    $trg_liniifact_upd_br$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_liniifact_upd_br BEFORE UPDATE ON liniifact

FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_upd_br();

Listingul 17.11 conţine ultimul dintre cele trei declanşatoare ale tabelei LINIIFACT – cel dedicat ştergerii unei linii.

Listingul 17.11. Noua versiune a declanşatorului de ştergere a unei înregistrări din LINIIFACT

```
DROP FUNCTION trg_liniifact_del() CASCADE;

DROP FUNCTION trg_liniifact_del2() CASCADE;
CREATE OR REPLACE FUNCTION trg_liniifact_del2()
    RETURNS TRIGGER AS $trg_liniifact_del2$
BEGIN
    -- noutate !!!
    PERFORM p_init_vp();
    PERFORM p_set ('v_trg_liniifact', TRUE);
    --
    UPDATE facturi
        SET tva = tva - OLD.tvalinie,
          valtotala = valtotala - OLD.cantitate * OLD.pretunit - OLD.tvalinie
        WHERE nrfact = OLD.nrfact;

    PERFORM p_set ('v_trg_liniifact', FALSE);
    RETURN OLD;
END;
$trg_liniifact_del2$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_liniifact_del2 BEFORE DELETE ON liniifact

FOR EACH ROW

EXECUTE PROCEDURE trg_liniifact_del2();

Cele trei declanşatoare asigură semnalizarea utilă declanşatorului de modificare în FACTURI, declanşator care trebuie să „discearnă" o modificare bună de una rea (vezi listingul 17.12).

Listingul 17.12. Declanşatorul de modificare a unei înregistrări din FACTURI

```
DROP FUNCTION trg_facturi_upd() CASCADE;

CREATE OR REPLACE FUNCTION trg_facturi_upd()
    RETURNS TRIGGER AS $trg_facturi_upd$
```

```
BEGIN
    IF NEW.tva <> OLD.tva OR NEW.valtotala <> OLD.valtotala THEN
        PERFORM p_init_vp();
        IF NOT f_trg_liniifact ()THEN
                RAISE EXCEPTION
                    'Nu puteti modifica interactiv nici valoarea facturii, nici TVA-ul acesteia!';
        END IF;
    END IF;
    RETURN NEW;
END;
$trg_facturi_upd$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_facturi AFTER UPDATE ON facturi FOR EACH ROW

EXECUTE PROCEDURE trg_facturi_upd();

După atâta străduinţă, a sosit momentul verificărilor. Să începem printr-o tentativă frauduloasă de a modifica direct valoarea atributului ValTotală:

UPDATE facturi SET ValTotala = 12345678 WHERE NrFact = 1111

Spre meritul lor (şi al nostru) declanşatoarele îşi fac datoria, după cum sugerează figura 17.5.

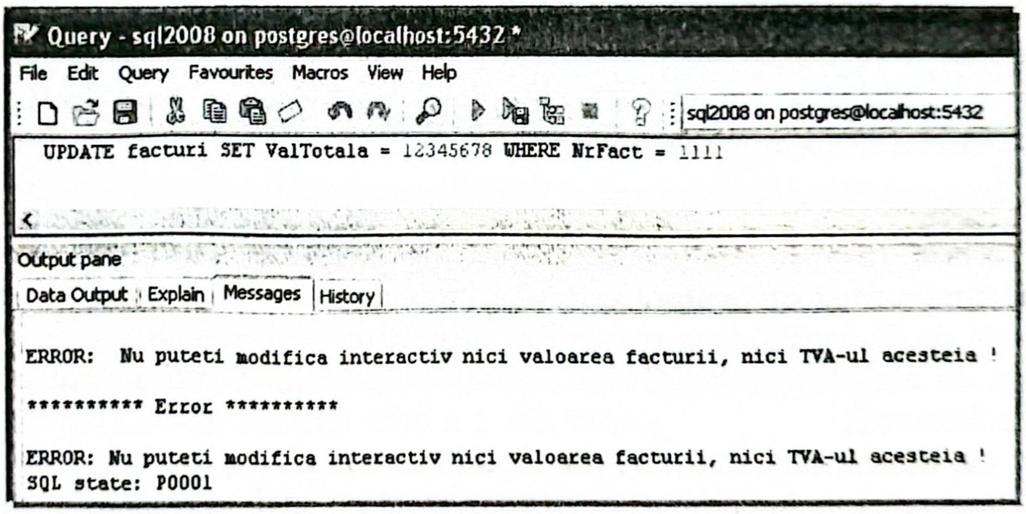


Figura 17.5. Blocarea unei modificări imorale în tabela FACTURI

În schimb, dacă ştergem liniile rămase în factura 1111 şi le readăugăm după scenariul din capitolul trei, lucrurile decurg fără incidente:

DELETE FROM liniifact WHERE NrFact = 1111;

INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)

VALUES (1111, 1, 1, 50, 1000);

INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)

VALUES (1111, 2, 2, 75, 1050);

INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)

VALUES (1111, 3, 5, 500, 7060);

17.3. Declanșatoare în PL/SQL

În materie de declanșatoare, limbajul PL/SQL este deopotrivă și unul dintre cele mai generoase și unul dintre cele mai năzuroase. La tipologia din tabelul 8.1 PL/SQL adaugă declanșatoarele pentru conectarea și închiderea conexiunii la/cu o bază de date, declanșatoare de pornire/oprire a bazei de date și declanșatoare asociate operațiunilor de tip DDL (CREATE/ALTER/DROP TABLE/PROCEDURE/FUNCTION/TRIGGER/... (vezi și [Fotache *et al.* 2003]).

Începem ca în paragraful precedent prin a crea o secvență – *seq_codcl*. Formatul comenzii este destul de apropiat celui din PostgreSQL:

```
CREATE SEQUENCE seq_codcl START WITH 1011 INCREMENT BY 1
        MINVALUE 1001 MAXVALUE 8900 NOCYCLE NOCACHE ORDER;
```

Cum e și normal, un declanșator se creează în PL/SQL direct prin comanda CREATE TRIGGER, iar structura acestuia o respectă pe cea a oricărui bloc PL/SQL, cu câteva deosebiri. Declanșatorul de tip BEFORE-INSERT-ROW pentru tabela CLIENȚI este cel din listingul 17.13.

Listingul 17.13. Declanșator PL/SQL pentru inserarea unei linii în tabela CLIENȚI

```
CREATE OR REPLACE TRIGGER trg_clienti_ins_before_row
 BEFORE INSERT ON clienti FOR EACH ROW
BEGIN
    :NEW.CodCl := seq_codcl.NEXTVAL;
END;
```

Extragerea următoarei valori unice disponibile din secvență se realizează ușor diferit față de PL/pgSQL: *seq_codcl.NEXTVAL*. Până la versiunea 11g, în PL/SQL comanda de atribuire a valorii din secvență atributului *CodCl* se putea realiza numai în formatul:

```
SELECT seq_codcl.NEXTVAL INTO :NEW.CodCl FROM dual
```

Oracle are alergie la opțiunea ON UPDATE CASCADE în comenzile CREATE TABLE și ALTER TABLE. Printr-un declanșator de modificare a unei linii putem remedia acest lucru. Notăm ca plus al PL/SQL-ului că un declanșator de modificare poate fi asociat, chiar la creare, numai unui atribut (sau unei părți dintre atributele) din tabelă. Spre exemplu, dorim ca orice modificare a numărului unei facturi să se propage automat în tabelele-copil, LINIIFACT și ÎNCASFACT. Declanșatorul care va rezolva această problemă este cel din listingul 17.14.

Listingul 17.14. Declanșator PL/SQL pentru substituirea opțiunii ON UPDATE CASCADE

```
CREATE OR REPLACE TRIGGER trg_facturi_upd_a_r_NrFact
  AFTER UPDATE OF NrFact ON facturi FOR EACH ROW
BEGIN
  UPDATE liniifact SET NrFact = :NEW.NrFact WHERE NrFact = :OLD.NrFact;
  UPDATE incasfact SET NrFact = :NEW.NrFact WHERE NrFact = :OLD.NrFact;
END;
```

Declanșatorul pentru inserarea de înregistrări în tabela FACTURI (listingul 17.15) este constuit de o manieră similară și nu ridică nicio problemă.

Listingul 17.15. Declanşatorul PL/SQL pentru inserarea unei înregistrări în FACTURI

```
CREATE OR REPLACE TRIGGER trg_facturi_ins
    BEFORE INSERT ON facturi FOR EACH ROW
BEGIN
    :NEW.valtotala := 0;
    :NEW.tva := 0;
    :NEW.valincasata := 0;
END;
```

Nici pentru declanşatorul *trg_liniifact_ins* dedicat inserării unei înregistrări în tabela LINIIFACT nu apar modificări spectaculoase (vezi listingul 17.16).

Listingul 17.16. Declanşatorul PL/SQL pentru inserarea unei înregistrări în LINIIFACT

```
CREATE OR REPLACE TRIGGER trg_liniifacturi_ins
  BEFORE INSERT ON liniifact FOR EACH ROW
BEGIN
    :NEW.tvalinie := COALESCE(:NEW.cantitate,0) * COALESCE(:NEW.pretunit,0) *
                f_proctva2(:NEW.codpr);
    SELECT COALESCE(MAX(Linie),0) + 1 INTO :NEW.linie FROM liniifact
    WHERE NrFact=:NEW.NrFact;

    UPDATE facturi SET tva = COALESCE(tva,0) + :NEW.tvalinie,
        valtotala = COALESCE(valtotala,0) + :NEW.cantitate * :NEW.pretunit + :NEW.tvalinie
     WHERE nrfact = :NEW.nrfact;
END;
```

Testarea cu aceleaşi trei comenzi INSERT din paragraful anterior a celor două declanşatoare este încununată de succes:

INSERT INTO facturi (nrfact, datafact, codcl, valtotala)

VALUES (3500, CURRENT_DATE, 1002, 33333333);

INSERT INTO liniifact VALUES (3500, 1, 2, 200, 100, 8989989);

INSERT INTO liniifact VALUES (3500, 2, 3, 100, 100, 0);

Necazurile încep odată cu tentativa de portare a declanşatorului din listingul 17.4 – *trg_liniifact_del*. Transcrierea din listingul 17.17 respectă aceeaşi idee, cu câteva diferenţe, cea mai vizibilă fiind înlocuirea comenzii RAISE EXCEPTION cu mai durdulia RAISE_APPLICATION_ERROR.

Listingul 17.17. Declanşatorul PL/SQL pentru ştergerea unei înregistrări din LINIIFACT

```
CREATE OR REPLACE TRIGGER trg_liniifact_del
    BEFORE DELETE ON liniifact FOR EACH ROW
DECLARE
    v_nrlinii liniifact.linie%TYPE;
BEGIN
    SELECT MAX(linie) INTO v_nrlinii FROM liniifact WHERE nrfact = :OLD.nrfact;
    IF :OLD.linie <> COALESCE(v_nrlinii,0) THEN
        RAISE_APPLICATION_ERROR(-20567,
                'Nu puteti sterge decat ultima linie dintr-o factura ');
    END IF;

    UPDATE facturi
        SET tva = tva - :OLD.tvalinie,
                valtotala = valtotala - :OLD.cantitate * :OLD.pretunit - :OLD.tvalinie
        WHERE nrfact = :OLD.nrfact;
END;
```

La compilare, PL/SQL se comportă paşnic, însă la verificarea sa prin comanda:

DELETE FROM liniifact WHERE NrFact = 3500 AND linie = 1

suntem luaţi pe nepusă masă de mesajul de eroare din figura 17.6. Ni se spune că tabela LINIIFACT este schimbătoare (ca şi omul, de altminteri). Din mesaj şi din documentaţie deducem că Oracle nu permite ca în declanşatoarele de modificare şi ştergere (şi cele de inserare, dacă se foloseşte sintaxa *INSERT INTO... SELECT...*) să apară o interogare (SELECT) chiar pe tabela în declanşatorul căreia suntem.

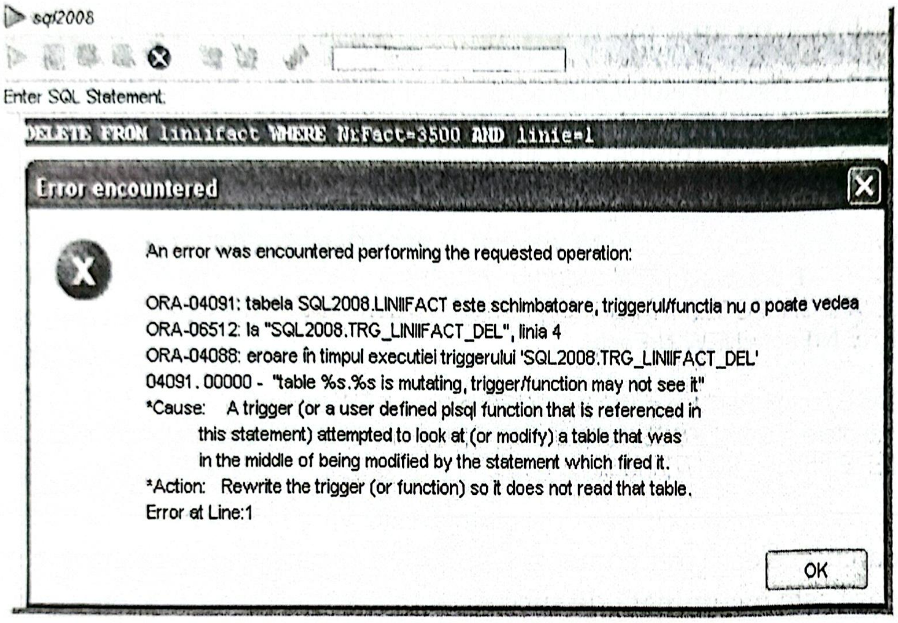


Figura 17.6. O problemă „clasică" în declanşatoarele de tip ROW în Oracle PL/SQL

Această restricţie este valabilă numai pentru declanşatoarele la nivel de linie. Aşa că ne-am putea gândi să evităm problema, pasând SELECT-ul în declanşatorul de tip AFTER STATEMENT, ultimul executat. Dar în declanşatoarele la nivel de comandă nu avem acces la valorile vechi (:OLD) şi noi (:NEW) de pe linia ştearsă sau modificată. O soluţie anemică este să folosim variabile publice pe care le „umplem" în declanşatorul la nivel de linie şi pe care le valorificăm în declanşatorul AFTER STATEMENT. Oricum, în ultimul paragraf din capitolul 16 am creat pachetul pac_vânzări în specificaţiile căruia apar, ca din întâmplare, patru variabile (publice) *v_trg_linii_fact*, *v_trg_încasfact*, *v_nrfact* şi *v_linie*.

Soluţia din listingul 17.18 este articulată în trei declanşatoare. Primul - *trg_liniifact_del1* - este de tip BEFORE-STATEMENT şi NULL-iează cele două variabile publice. Al doilea (în ordinea execuţiei) - *trg_liniifact_del2* - este de tip AFTER ROW şi, pe lângă decrementarea celor două atribute calculate din tabela FACTURI, stochează numărul facturii şi numărul liniei şterse în cele două variabile publice. Al treilea - *trg_liniifact_del3* - este ultimul executat (AFTER STATEMENT) şi face verificarea propriu-zisă.

Listingul 17.18. Trei declanşatoare PL/SQL pentru ştergerea unei înregistrări din LINIIFACT

```
DROP TRIGGER trg_liniifact_del;

-- primul declansator - BEFORE STATEMENT- "curata" variabilele publice
```

```
CREATE OR REPLACE TRIGGER trg_liniifact_del1 BEFORE DELETE ON liniifact
BEGIN
  pac_vanzari.v_nrfact := NULL;
  pac_vanzari.v_linie := NULL;
END;

-- al doilea declansator – AFTER ROW – "seteaza" variabilele publice
-- si actualizeeaza FACTURI.tva si FACTURI.ValTotala
CREATE OR REPLACE TRIGGER trg_liniifact_del2
  AFTER DELETE ON liniifact FOR EACH ROW
BEGIN
  pac_vanzari.v_nrfact := :OLD.NrFact;
  pac_vanzari.v_linie := :OLD.Linie;
  UPDATE facturi
      SET tva = tva - :OLD.tvalinie,
          valtotala = valtotala - :OLD.cantitate * :OLD.pretunit - :OLD.tvalinie
      WHERE nrfact = :OLD.nrfact;
END;

-- al treilea declansator - AFTER STATEMENT - face verificarea
CREATE OR REPLACE TRIGGER trg_liniifact_del2
  AFTER DELETE ON liniifact
DECLARE
      v_nrlinii liniifact.linie%TYPE;
BEGIN
  SELECT MAX(linie) INTO v_nrlinii FROM liniifact WHERE nrfact = pac_vanzari.v_nrfact;
      IF pac_vanzari.v_linie < COALESCE(v_nrlinii,0) THEN
          RAISE_APPLICATION_ERROR(-20567,
                      'Nu puteti sterge decat ultima linie dintr-o factura ');
      END IF;
END;
```

Acum comanda de ştergere funcţionează. Cu o mică observaţie, esenţială! Ce se întâmplă dacă am şterge mai multe linii cu o singură comanda DELETE? Comanda:

DELETE FROM liniifact WHERE NrFact = 3500 AND Linie = 1 OR

NrFact = 3119 AND linie IN (2,3)

ar trebui blocată, întrucât din factura 3500 se şterge prima linie, iar factura are două. Comanda, însă, se execută fără obiecţii, ceea ce, evident, este o eroare. Explicaţia ţine de faptul că declanşatoarele păstrează numai ultima linie din ultima factură din care a avut loc ştergerea.

Prin urmare, în loc de variabile simple, va trebui să folosim colecţii. Dintre cele trei tipuri de masive/colecţii disponibile în PL/SQL vor recurge la vectori asociativi, utilizabili doar în programe (nu şi în tabele). Listingul 17.19 conţine specificaţiile pachetului *pac_vânzări* în care definim două tipuri de vectori asociativi – *t_nrfacturi* şi *t_linii* – şi două variabile de acest tip – *v_nrfacturi* şi *v_linii*.

Listingul 17.19. Două variabile masiv (publice) în PL/SQL

```
CREATE OR REPLACE PACKAGE pac_vanzari AS
    -- variabile publice
    v_trg_liniifact BOOLEAN := FALSE;
    v_trg_incasfact BOOLEAN := FALSE;
    v_unde_sunt VARCHAR2(30);
```

```
    TYPE t_nrfacturi IS TABLE OF liniifact.nrfact%TYPE INDEX BY PLS_INTEGER;
    TYPE t_linii IS TABLE OF liniifact.linie%TYPE INDEX BY PLS_INTEGER;
    v_nrfacturi t_nrfacturi;
    v_linii t_linii;

    FUNCTION f_proctva (codpr_ produse.codpr%TYPE) RETURN produse.proctva%TYPE;
    FUNCTION f_proctva (denpr_ produse.denpr%TYPE) RETURN produse.proctva%TYPE;
    FUNCTION f_valtot (nrfact_ facturi.nrfact%TYPE)    RETURN NUMERIC;
    FUNCTION f_val_cumul (nrfact_ facturi.nrfact%TYPE) RETURN NUMERIC;
    PROCEDURE p_cumul;
    PROCEDURE p_cumul2;
END; -- pac_vanzari
/
```

Cele trei declanşatoare trebuie rescrise ca în listingul 17.20. Cel de-al treilea devine un pic mai corpolent întrucât este nevoie de o buclă care să parcurgă toate componentele celor două variabile-colecţie.

Listingul 17.20. Noile versiuni ale celor trei declanşatoare de ştergere în LINIIFACT ce utilizează variabilele masiv

```
-- refacem primul declansator - BEFORE STATEMENT
CREATE OR REPLACE TRIGGER trg_liniifact_del1 BEFORE DELETE ON liniifact
BEGIN
  pac_vanzari.v_nrfacturi.DELETE;
  pac_vanzari.v_linii.DELETE;
END;

-- al doilea declansator -- AFTER ROW -- rescris
CREATE OR REPLACE TRIGGER trg_liniifact_del2
  BEFORE DELETE ON liniifact FOR EACH ROW
BEGIN
  pac_vanzari.v_nrfacturi (pac_vanzari.v_nrfacturi.COUNT + 1) := :OLD.NrFact;
  pac_vanzari.v_linii (pac_vanzari.v_linii.COUNT + 1) := :OLD.Linie;

  UPDATE facturi
     SET tva = tva - :OLD.tvalinie,
         valtotala = valtotala - :OLD.cantitate * :OLD.pretunit - :OLD.tvalinie
     WHERE nrfact = :OLD.nrfact;
END;

-- al treilea declansator -- AFTER STATEMENT -- face verificarea
CREATE OR REPLACE TRIGGER trg_liniifact_del3
  AFTER DELETE ON liniifact
DECLARE
     v_nrlinii liniifact.linie%TYPE := 0;
BEGIN
  FOR i IN 1..pac_vanzari.v_nrfacturi.COUNT LOOP
    SELECT MAX(linie) INTO v_nrlinii FROM liniifact WHERE nrfact = pac_vanzari.v_nrfacturi(i);
    IF pac_vanzari.v_linii(i) < COALESCE(v_nrlinii,0) THEN
      RAISE_APPLICATION_ERROR(-20567,
      'Nu puteti sterge decat ultima linie dintr-o factura ');
    END IF;
  END LOOP;
END;
```

În noua versiune, cele trei declanşatoare par a-şi face bine treaba. Dar, după cum meditam în precedentul paragraf, această soluţie este mult prea rigidă pentru

a fi acceptabilă, aşa că ne propunem să redactăm varianta după care în facturile în care s-a şters măcar o linie se numerotează automat liniile rămase. Ca să nu mai umblăm în corpul pachetului *pac_vânzări* (ar trebui să compilăm toate funcţiile şi procedurile deja existente), creăm unul nou-nouţ pe care îl vom denumi *pac_colecţii* (vezi listingul 17.21).

Listingul 17.21. Specificaţiile şi corpul pachetului ***pac_colecţii***

```
CREATE OR REPLACE PACKAGE pac_colectii AS
  TYPE t_nrfacturi IS TABLE OF liniifact.nrfact%TYPE INDEX BY PLS_INTEGER;
  TYPE t_linii IS TABLE OF liniifact.linie%TYPE INDEX BY PLS_INTEGER;
  v_nrfacturi t_nrfacturi;
  v_linii t_linii;
  FUNCTION f_apare_in_nrfacturi (nrfact_ liniifact.nrfact%TYPE) RETURN BOOLEAN;
END; -- pac_colectii
/

CREATE OR REPLACE PACKAGE BODY pac_colectii AS
------------------------------------------------------------------------
 FUNCTION f_apare_in_nrfacturi (nrfact_ liniifact.nrfact%TYPE) RETURN BOOLEAN
 IS
  v_este BOOLEAN := FALSE;
 BEGIN
  FOR i IN 1..v_nrfacturi.COUNT LOOP
   IF v_nrfacturi(i) = nrfact_ THEN
    v_este := TRUE;
    EXIT;
   END IF;
  END LOOP;
  RETURN v_este;
 END f_apare_in_nrfacturi;

END; -- pac_colectii
/
```

Pe lângă cele două tipuri şi variabile vectori asociativi, am creat şi o funcţie de căutare a unui număr de factură în vectorul *v_nrfacturi*. Dintre cele trei declanşatoare din listingul 17.22, ultimul este cel mai diferit faţă de varianta anterioară, fiind cel care renumerotează toate liniile în facturile în care avem măcar o linie ştearsă.

Listingul 17.22. (Şi) mai noile declanşatoare de ştergere în LINIIFACT

```
-- primul declansator – BEFORE-STATEMENT
CREATE OR REPLACE TRIGGER trg_liniifact_del1 BEFORE DELETE ON liniifact
BEGIN
 pac_colectii.v_nrfacturi.DELETE;
 pac_colectii.v_linii.DELETE;
END;
/
------------------------------------------------------------------------
-- al doilea declansator - AFTER-ROW
CREATE OR REPLACE TRIGGER trg_liniifact_del2
 BEFORE DELETE ON liniifact FOR EACH ROW
BEGIN
 IF pac_colectii.f_apare_in_nrfacturi(:OLD.NrFact) = FALSE THEN
  pac_colectii.v_nrfacturi (pac_colectii.v_nrfacturi.COUNT + 1) := :OLD.NrFact;
```

```
   END IF;

   UPDATE facturi
      SET tva = tva - :OLD.tvalinie,
          valtotala = valtotala - :OLD.cantitate * :OLD.pretunit - :OLD.tvalinie
      WHERE nrfact = :OLD.nrfact;
END;
/
--------------------------------------------------------------------------------
-- al treilea declansator - AFTER-STATEMENT
CREATE OR REPLACE TRIGGER trg_liniifact_del3
  AFTER DELETE ON liniifact
DECLARE
  j NUMBER(2);
  CURSOR c_liniifact (nrfact_ liniifact.nrfact%TYPE) IS
     SELECT * FROM liniifact WHERE NrFact = nrfact_ ORDER BY Linie FOR UPDATE;
BEGIN
  FOR i IN 1..pac_colectii.v_nrfacturi.COUNT LOOP
    j := 1;
    FOR rec_liniifact IN c_liniifact (pac_colectii.v_nrfacturi(i)) LOOP
      UPDATE liniifact SET linie = j WHERE CURRENT OF c_liniifact;
      j := j + 1;
    END LOOP;
  END LOOP;
END;
/
```

Figura 17.7 ilustrează situaţia liniilor facturii 3119 înainte şi după ştergerea celor cu numerele 1 şi 3. Totul pare a fi în regulă. Atenţie, însă! Declanşatorul *trg_liniifact_del3* va modifica în unele înregistrări din LINIIFACT numărul liniei, ceea ce poate atrage lansarea eventualului declanşator de modificare a aceleiaşi tabele.

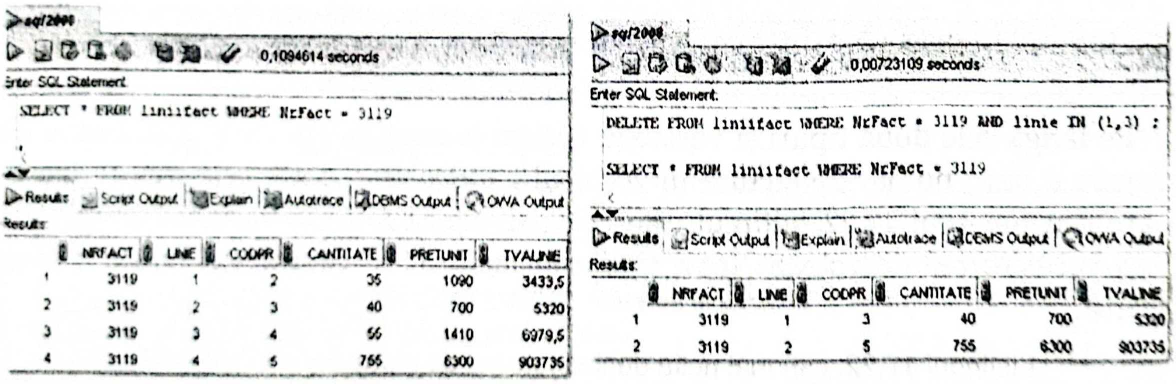


Figura 17.7. Renumerotarea liniilor prin declanşatoare în factura 3119

Ne propunem să rezolvăm şi problema protejării atributelor calculate de modificările neautorizate. Pentru aceasta vom pune la treabă o altă variabilă publică din pachetul *pac_vânzări* (listingul 17.19) – *v_trg_liniiifact*. Această variabilă va avea valoarea logică TRUE doar pe parcursul execuţiei declanşatoarelor de inserare, modificare şi ştergere ale tabelei LINIIFACT. Prin urmare, în trei dintre declanşatoare – *trg_liniifact_ins*, *trg_liniifact_del2* şi *trg_liniifact_upd_br* – variabilă va fi setată pe TRUE imediat înainte de *UPDATE facturi* ... (comanda care modifică valorile atributelor ValTotală şi TVA) şi modificată pe FALSE imediat după UPDATE.

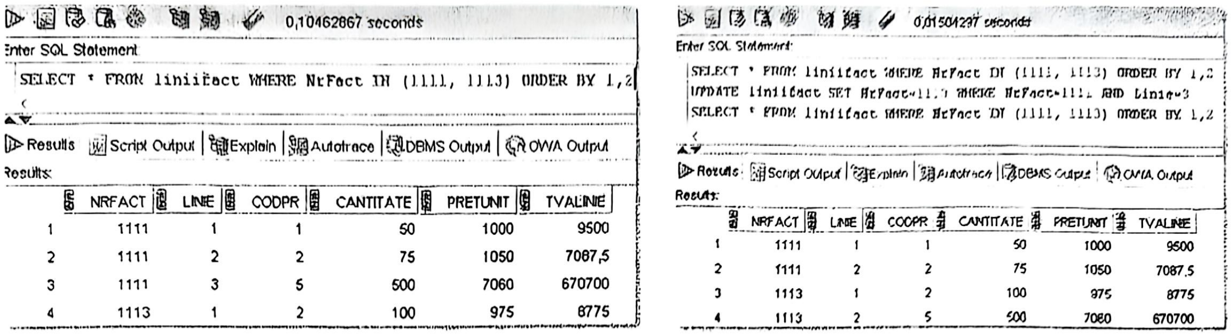
Listingul 17.23. Noile declanşatoare ale LINIIFACT care asigură „protecţia" atributelor calculate din FACTURI

```
CREATE OR REPLACE TRIGGER trg_liniifact_ins
  BEFORE INSERT ON liniifact FOR EACH ROW
BEGIN
  pac_vanzari.v_trg_liniifact := TRUE;
     :NEW.tvalinie := COALESCE(:NEW.cantitate,0) * COALESCE(:NEW.pretunit,0) *
                    f_proctva2(:NEW.codpr);
     SELECT COALESCE(MAX(Linie),0) + 1 INTO :NEW.linie FROM liniifact
     WHERE NrFact=:NEW.NrFact;

     UPDATE facturi SET tva = COALESCE(tva,0) + :NEW.tvalinie,
         valtotala = COALESCE(valtotala,0) + :NEW.cantitate * :NEW.pretunit + :NEW.tvalinie
      WHERE nrfact = :NEW.nrfact;
  pac_vanzari.v_trg_liniifact := FALSE;
END;
/
-----------------------------------------------------------------------------------
-- se modifica al doilea declansator de stergere – AFTER-ROW
CREATE OR REPLACE TRIGGER trg_liniifact_del2
  BEFORE DELETE ON liniifact FOR EACH ROW
BEGIN
  IF pac_colectii.f_apare_in_nrfacturi(:OLD.NrFact) = FALSE THEN
   pac_colectii.v_nrfacturi (pac_colectii.v_nrfacturi.COUNT + 1) := :OLD.NrFact;
  END IF;
  pac_vanzari.v_trg_liniifact := TRUE;
  UPDATE facturi
     SET tva = tva - :OLD.tvalinie,
         valtotala = valtotala - :OLD.cantitate * :OLD.pretunit - :OLD.tvalinie
     WHERE nrfact = :OLD.nrfact;
  pac_vanzari.v_trg_liniifact := FALSE;
END;
/
-----------------------------------------------------------------------------------
– modificam usor si al treilea declansator de stergere
CREATE OR REPLACE TRIGGER trg_liniifact_del3
  AFTER DELETE ON liniifact
DECLARE
  j NUMBER(2);
  CURSOR c_liniifact (nrfact_ liniifact.nrfact%TYPE) IS
     SELECT * FROM liniifact WHERE NrFact = nrfact_ ORDER BY Linie FOR UPDATE;
BEGIN
  FOR i IN 1..pac_colectii.v_nrfacturi.COUNT LOOP
    j := 1;
    FOR rec_liniifact IN c_liniifact (pac_colectii.v_nrfacturi(i)) LOOP
      pac_vanzari.v_unde_sunt := 'trg_liniifact_del3';
      UPDATE liniifact SET linie = j WHERE CURRENT OF c_liniifact;
      pac_vanzari.v_unde_sunt := NULL;
      j := j + 1;
    END LOOP;
  END LOOP;
END;
/
-----------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER trg_liniifact_upd_bs
  BEFORE UPDATE OF NrFact, CodPr, Cantitate, PretUnit, TVALinie ON liniifact
BEGIN
  IF COALESCE(pac_vanzari.v_unde_sunt, ' ') NOT IN
```

```
    ( 'trg_liniifact_upd_as', 'trg_liniifact_del3') THEN
    pac_colectii.v_nrfacturi.DELETE;
  END IF;
END;
/
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER trg_liniifact_upd_br_linie
  BEFORE UPDATE OF Linie ON liniifact FOR EACH ROW
BEGIN
  IF COALESCE(pac_vanzari.v_unde_sunt, ' ') NOT IN
        ('trg_liniifact_del3', 'trg_liniifact_upd_as') THEN
    RAISE_APPLICATION_ERROR(-20456,
    'Nu aveti permisiunea de a modifica valoarea atributului Linie !');
  END IF;
END;
/
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER trg_liniifact_upd_br
  BEFORE UPDATE OF NrFact, CodPr, Cantitate, PretUnit, TVALinie ON liniifact
        FOR EACH ROW
BEGIN
    pac_vanzari.v_unde_sunt := 'trg_liniifact_upd_br';
    pac_vanzari.v_trg_liniifact := TRUE;
    :NEW.tvalinie := COALESCE(:NEW.cantitate,0) * COALESCE(:NEW.pretunit,0) *
          f_proctva2(:NEW.codpr);
     UPDATE facturi SET tva = COALESCE(tva,0) - :OLD.tvalinie,
        valtotala = COALESCE(valtotala,0) - :OLD.cantitate * :OLD.pretunit + :OLD.tvalinie
    WHERE nrfact = :OLD.nrfact;

     UPDATE facturi SET tva = COALESCE(tva,0) + :NEW.tvalinie,
        valtotala = COALESCE(valtotala,0) + :NEW.cantitate * :NEW.pretunit + :NEW.tvalinie
    WHERE nrfact = :NEW.nrfact;

    IF :NEW.NrFact <> :OLD.NrFact THEN
     pac_colectii.v_nrfacturi(pac_colectii.v_nrfacturi.COUNT+1) := :OLD.NrFact;
     pac_colectii.v_nrfacturi(pac_colectii.v_nrfacturi.COUNT+1) := :NEW.NrFact;
    END IF;
    pac_vanzari.v_trg_liniifact := FALSE;
    pac_vanzari.v_unde_sunt := NULL;
END;
/
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER trg_liniifact_upd_as
  AFTER UPDATE OF NrFact, CodPr, Cantitate, PretUnit, TVALinie ON liniifact
DECLARE
  j NUMBER(2);
  CURSOR c_liniifact (nrfact_ liniifact.nrfact%TYPE) IS
     SELECT * FROM liniifact WHERE NrFact = nrfact_ ORDER BY Linie FOR UPDATE;
BEGIN
  IF COALESCE(pac_vanzari.v_unde_sunt, ' ') IN
     ( 'trg_liniifact_upd_as', 'trg_liniifact_del3')  THEN
    NULL;
  ELSE
    FOR i IN 1..pac_colectii.v_nrfacturi.COUNT LOOP
     j := 1;
     FOR rec_liniifact IN c_liniifact (pac_colectii.v_nrfacturi(i)) LOOP
       pac_vanzari.v_unde_sunt := 'trg_liniifact_upd_as';
       UPDATE liniifact SET linie = j WHERE CURRENT OF c_liniifact;
       pac_vanzari.v_unde_sunt := NULL;
```

```
        i := i + 1;
      END LOOP;
    END LOOP;
  END IF;
END;
/
```

Înainte de a trece la redactarea declanşatorului de „protecţie", punem la încercare aceste declanşatoare printr-o comandă UPDATE care să modifice, pentru o înregistrare din LINIIFACT, numărul facturii:



	NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	TVALINIE
1	1111	1	1	50	1000	9500
2	1111	2	2	75	1050	7087,5
3	1111	3	5	500	7060	670700
4	1113	1	2	100	975	8775

	NRFACT	LINIE	CODPR	CANTITATE	PRETUNIT	TVALINIE
1	1111	1	1	50	1000	9500
2	1111	2	2	75	1050	7087,5
3	1113	1	2	100	975	8775
4	1113	2	5	500	7060	670700

Figura 17.8. Trecerea unei linii din factura 1111 în factura 1113

SELECT * FROM liniifact WHERE NrFact IN (1111, 1113) ORDER BY 1,2;

UPDATE liniifact SET NrFact = 1113 WHERE NrFact = 1111 AND Linie = 3;

SELECT * FROM liniifact WHERE NrFact IN (1111, 1113) ORDER BY 1,2;

Figura 17.8, ce suprinde liniile facturilor 1111 şi 1113 înainte şi după modificare, creşte gradul de încredere în declanşatoarele tabelei LINIIFACT.

Nu ne-a rămas decât să redactăm declanşatorul de modificare a celor două atribute calculate din tabela FACTURI, declanşator care va testa valoarea variabilei *v_trg_liniifact* din pachetul *pac_vanzari* - vezi listingul 17.24.

Listingul 17.24. Declanşatorul care blochează modificarea directă a atributelor calculate

```
CREATE OR REPLACE TRIGGER trg_facturi_upd_br1
  BEFORE UPDATE OF tva, valtotala ON facturi FOR EACH ROW
BEGIN
    IF pac_vanzari.v_trg_liniifact THEN
        -- e in regula, modificarile vin din declansatoarele LINIIFACT
        NULL;
   ELSE
      RAISE_APPLICATION_ERROR(-26789,
          'Nu aveti permisiunea de a modifica interactiv atributele TVA si ValTotala');
   END IF;
 END;
/
```

Figura 17.9 confirmă funcţionarea declanşatoarelor celor două tabele. Încercarea de modificare directă a valorii atributului TVA lansează *trigger*-ul *trg_facturi_upd_br1*. Întrucât, comanda UPDATE lasă „reci" declanşatoarele tabelei LINIIFACT variabila publică rămâne pe starea sa obişnuită - FALSE. Declanşatorul nu are încotro şi va genera mesajul de eroare.

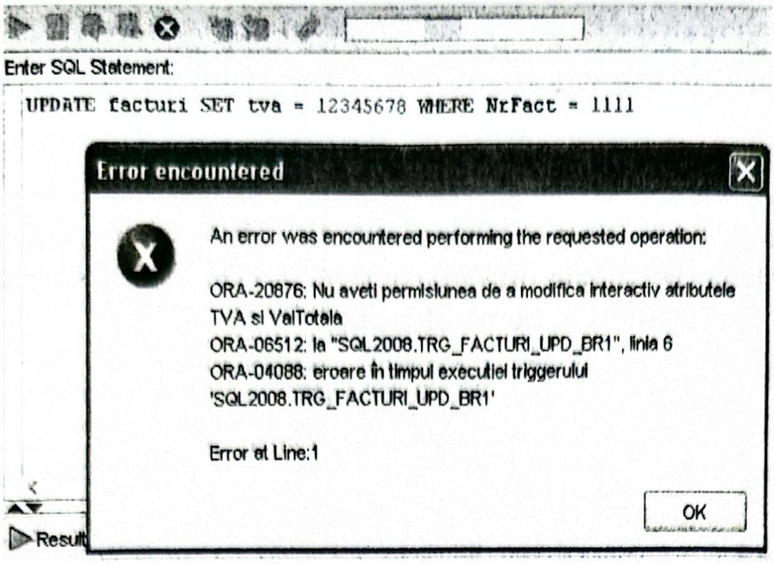


Figura 17.9. Tentativă blocată de modificare directă a TVA-ului unei facturi

Paragraful 14.3 făcea trimitere la problemele de actualizare a tabelelor-sursă la modificarea conţinutului tabelelor virtuale. Spuneam atunci că soluţia elegantă în aceste situaţii o reprezintă declanşatoarele de tip INSTEAD OF. Pentru a exemplifica un asemenea gen de declanşator, ne vom folosi de exemplul din paragraful 14.3.3:

```
CREATE VIEW vFacturi AS
        SELECT lf.NrFact, DataFact, DenCl, f.CodCl, Linie, lf.CodPr, DenPr,
                UM, ProcTVA, Grupa,
                Cantitate, PretUnit,
Cantitate * PretUnit AS ValFaraTVALinie,
                Cantitate * PretUnit * ProcTVA AS TVALinie,
                Cantitate * PretUnit * (1+ProcTVA) AS ValTotLinie
        FROM liniifact lf
                INNER JOIN produse p ON lf.CodPr = p.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
                INNER JOIN clienti c ON f.CodCl = c.CodCl
        ORDER BY lf.NrFact, Linie;
```

Ca element pregătitor, grupăm două funcţii de căutare într-un pachet special pe care-l vom numi *pac_există* (vezi listingul 17.25). Prima, *f_există_produs*, returnează TRUE dacă există în tabela PRODUSE un sortiment cu codul specificat, iar a doua, *f_există_factura*, returnează valoarea logică TRUE dacă există o factură (în FACTURI) cu numărul specificat la invocare.

Listingul 17.25. Un pachet nou

```
CREATE OR REPLACE PACKAGE pac_exista AS
  FUNCTION f_exista_produs (codpr_ produse.codpr%TYPE) RETURN BOOLEAN;
  FUNCTION f_exista_factura (nrfact_ facturi.nrfact%TYPE) RETURN BOOLEAN;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY pac_exista AS
 ---------------------------------------------------------------------------
 FUNCTION f_exista_produs (codpr_ produse.codpr%TYPE)
         RETURN BOOLEAN IS
  v_unu NUMBER(1);
 BEGIN
  SELECT 1 INTO v_unu FROM produse WHERE CodPr = codpr_;
  RETURN TRUE;
 EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RETURN FALSE;
 END f_exista_produs;
 ---------------------------------------------------------------------------
 FUNCTION f_exista_factura (nrfact_ facturi.nrfact%TYPE)
         RETURN BOOLEAN IS
   v_unu NUMBER(1);
 BEGIN
  SELECT 1 INTO v_unu FROM facturi WHERE NrFact = nrfact_;
  RETURN TRUE;
 EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RETURN FALSE;
 END  f_exista_factura;
END;
/
```

Declanşatorul executat la inserarea unei linii în tabela virtuală (vezi listingul 17.26) testează, înainte de adăugarea înregistrării în tabela LINIIFACT, dacă înregistrarea respectivă nu se referă la un produs nou (PRODUSE) şi/sau la o factură nouă (FACTURI).

Listingul 17.26. Declanşatorul INSTEAD OF INSERT

```
CREATE OR REPLACE TRIGGER trg_vFacturi_ioi INSTEAD OF INSERT ON vFacturi
BEGIN
 IF NOT pac_exista.f_exista_factura(:NEW.NrFact) THEN
    INSERT INTO facturi (NrFact, DataFact, CodCl)
        VALUES (:NEW.NrFact, :NEW.DataFact, :NEW.CodCl);
 END IF;
 IF NOT pac_exista.f_exista_produs(:NEW.CodPr) THEN
   INSERT INTO produse
        VALUES (:NEW.CodPr, :NEW.DenPr, :NEW.UM, :NEW.Grupa, :NEW.ProcTVA );
 END IF;
    INSERT INTO liniifact
        VALUES (:NEW.NrFact, :NEW.Linie, :NEW.CodPr, :NEW.Cantitate,
              :NEW.PretUnit, NULL );
END;
```

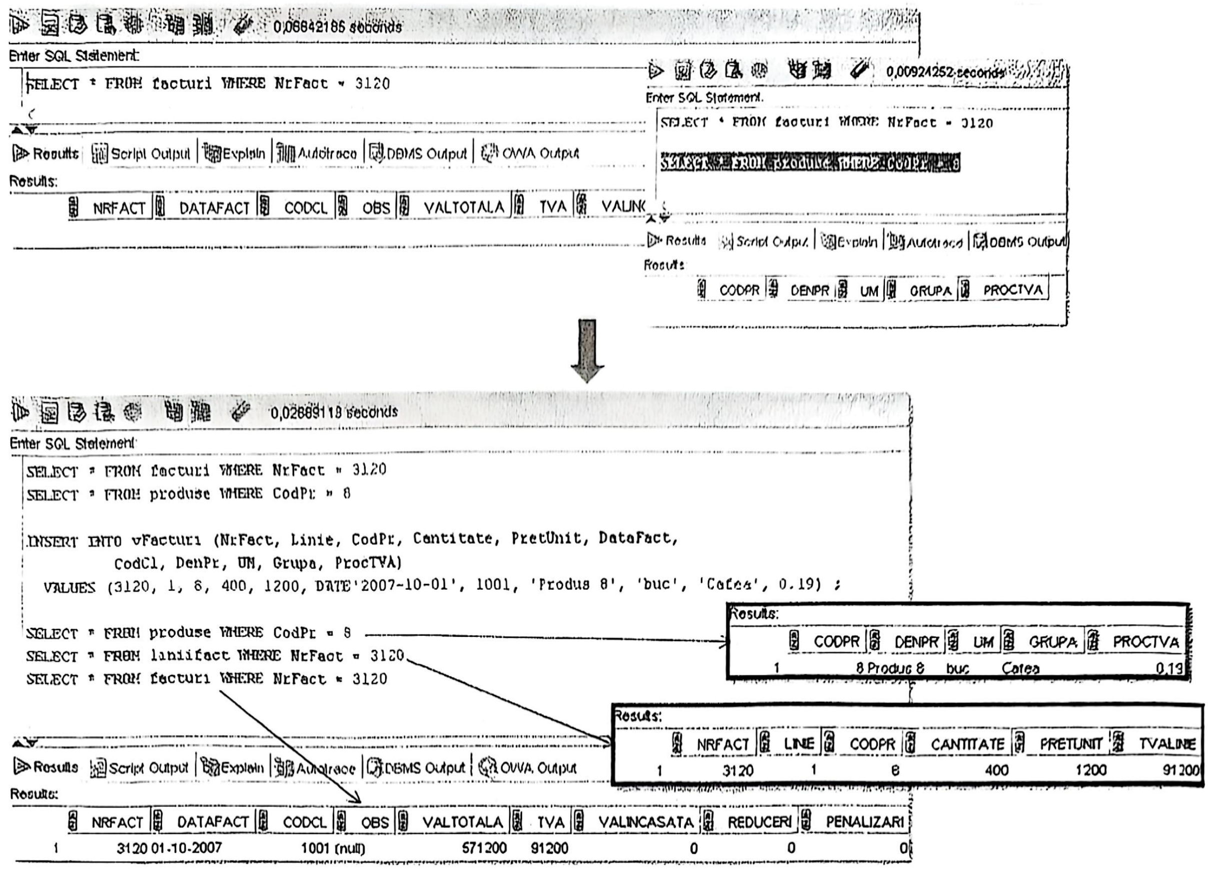


Figura 17.10. Verificarea funcţionării declanşatorului INSTEAD OF INSERT

Analog, pot fi create declanşatoare pentru modificări şi ştergeri ale liniilor din tabela virtuală. Verificarea funcţionării declanşatorului (figura 17.10) se face executând pe rând următoarele comenzi:

```
SELECT * FROM facturi WHERE NrFact = 3120
SELECT * FROM produse WHERE CodPr = 8

INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
                      CodCl, DenPr, UM, Grupa, ProcTVA)
VALUES (3120, 1, 8, 400, 1200, DATE'2007-10-01', 1001, 'Produs 8', 'buc',
        'Cafea', 0.19);

SELECT * FROM produse WHERE CodPr = 8
SELECT * FROM liniifact WHERE NrFact = 3120
SELECT * FROM facturi WHERE NrFact = 3120
```

17.4. Declanşatoare în Transact-SQL

Şi în T-SQL există atât declanşatoare de tip DML, cât şi de tip DDL (CREATE/ALTER/DROP). Spre deosebire de PL/pgSQL şi PL/SQL, tipologia *trigger*-elor DML este ceva mai săracă. Pentru o tabelă se pot defini numai declanşatoare de tip

AFTER şi INSTEAD OF la nivel de comenzi, iar pentru tabele virtuale declanşatoare de tip INSTEAD OF[4]. Să începem discuţia cu un caz simplu, cel al *trigger*-ului pentru inserarea de înregistrări în tabela FACTURI (vezi listingul 17.27).

Listingul 17.27. Prima versiune a declanşatorului INSERT pentru tabela FACTURI

```
CREATE TRIGGER trg_facturi_ins ON facturi FOR insert
AS
    UPDATE facturi SET ValTotala = 0, TVA = 0, ValIncasata = 0
    WHERE NrFact = (SELECT NrFact FROM inserted)
```

Poate cea mai importantă deosebire T-SQL ţine de faptul că aici declanşatoarele nu sunt la nivel de linie, ci la nivel de comandă. De aceea, nu avem posibilitatea de a folosi opţiunile OLD şi NEW pentru vechile şi noile valori. Există însă două tabele virtuale în care SQL Server stochează toate liniile care ne interesează. Prima tabelă logică (după titulatura documentaţiei SQL Server) virtuală este INSERTED şi conţine toate liniile inserate, sau - atenţie - liniile modificate (valorile noi rezultate după modificare). A doua este DELETED şi conţine liniile şterse sau liniile vechi (dinaintea modificărilor). Prin urmare, tabela INSERTED poate fi folosită la inserări şi modificări, iar DELETED la ştergeri şi modificări.

Testăm funcţionabilitatea declanşatorului:

```
INSERT INTO facturi VALUES (3500, GETDATE(), 1002, NULL, 33333,
        NULL, NULL, NULL, NULL)
SELECT * FROM facturi WHERE NrFact = 3500
```

Figura 17.11 conţine linia inserată în FACTURI prin comanda INSERT anterioară şi confirmă funcţionarea acestei prime versiuni a *trg_facturi_ins*.

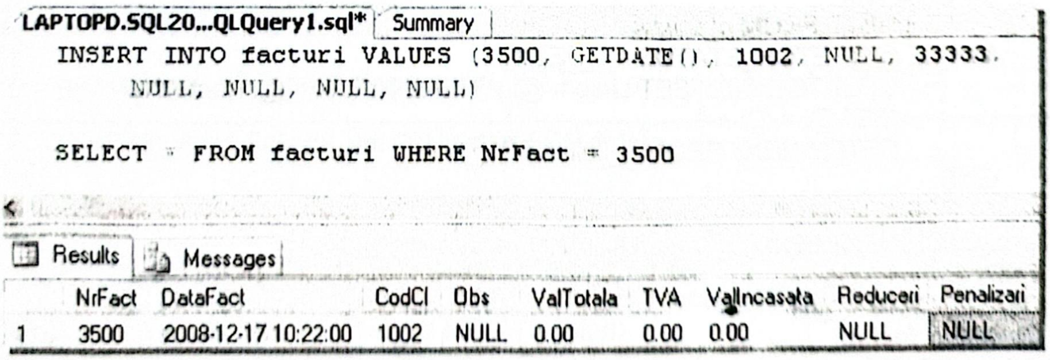


Figura 17.11. Verificarea funcţionării declanşatorului INSERT pentru tabela FACTURI

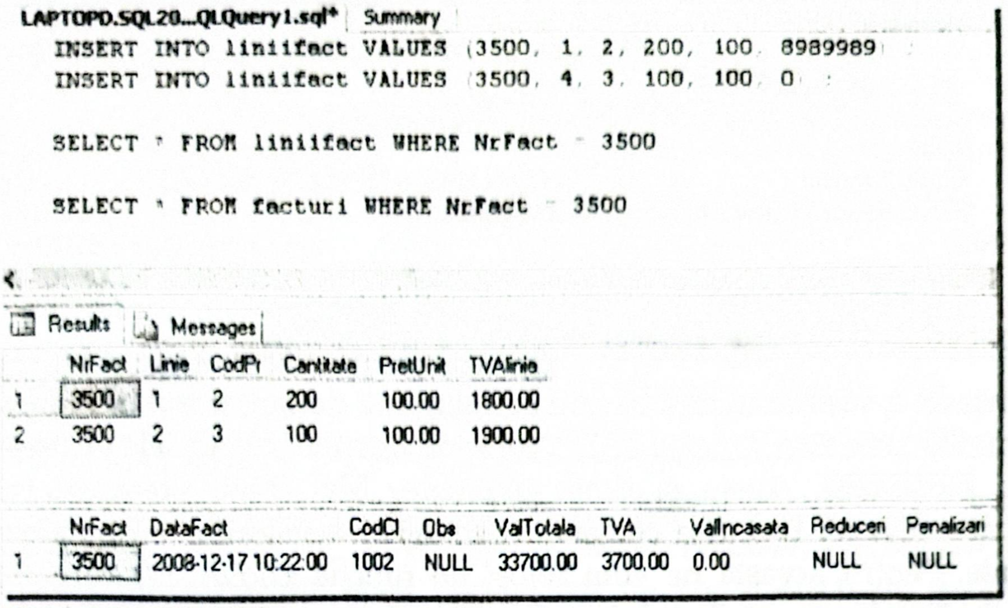
Continuăm discuţia cu declanşatorul de inserare (de înregistrări) în tabela LINIIFACT - *trg_liniifact_ins* - prezentat în listingul 17.28. După tipicul paragrafelor anterioare, acest declanşator trebuie să calculeze valoarea atributului TVAlinie pentru fiecare înregistrare inserată, să crească valoarea totală şi TVA-ul facturii în care se inserează fiecare linie şi, în plus, să aibă grijă ca liniile inserate în fiecare factură să fie numerotate consecutiv.

4. Ne interesează doar declanşatoarele T-SQL. Potrivit documentaţiei SQL Server, pot fi create şi alte *trigger*-e, sub forma unor metode implementate în .NET.

Listingul 17.28. Declanşatorul INSERT pentru tabela LINIIFACT

```
CREATE TRIGGER trg_liniifact_ins ON liniifact FOR insert
AS
    UPDATE liniifact
    SET TVAlinie = (        SELECT COALESCE(Cantitate,0) * COALESCE(PretUnit,0) *
                                    dbo.f_proctva(CodPr)
                            FROM inserted WHERE NrFact = liniifact.NrFact AND Linie = liniifact.Linie)
    WHERE CAST(NrFact AS CHAR(8)) + CAST(Linie AS CHAR(2)) =
          (SELECT CAST(NrFact AS CHAR(8)) + CAST (Linie AS CHAR(2))
           FROM inserted)

    UPDATE facturi
    SET TVA = (SELECT SUM(TVAlinie) FROM liniifact WHERE NrFact = facturi.NrFact),
        ValTotala = (SELECT SUM(Cantitate * PretUnit + TVAlinie) FROM liniifact
                                    WHERE NrFact = facturi.NrFact)
    WHERE NrFact IN (SELECT NrFact FROM inserted)

    DECLARE @i TINYINT;
    DECLARE @linie TINYINT;
    DECLARE @nrfact INT;

    DECLARE c_facturi CURSOR FOR
        SELECT DISTINCT NrFact FROM inserted ORDER BY NrFact
    OPEN c_facturi
    FETCH NEXT FROM c_facturi INTO @nrfact
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @i = (SELECT COALESCE(MAX(Linie),0) FROM liniifact
                        WHERE NrFact = @nrfact AND Linie NOT IN (
                                SELECT Linie FROM inserted WHERE NrFact = @nrfact)
                        ) + 1
        DECLARE c_linii CURSOR FOR
                SELECT Linie FROM inserted WHERE NrFact = @nrfact ORDER BY Linie
        OPEN c_linii
        FETCH NEXT FROM c_linii INTO @linie
        WHILE @@FETCH_STATUS = 0 BEGIN
                UPDATE liniifact SET Linie = @i WHERE NrFact = @nrfact AND Linie = @linie
                SET @i = @i + 1
                FETCH NEXT FROM c_linii INTO @linie
        END
        CLOSE c_linii
        FETCH NEXT FROM c_facturi INTO @nrfact
    END
    DEALLOCATE c_linii
    CLOSE c_facturi
    DEALLOCATE c_facturi
```

Am ţinut cont de faptul că tabela virtuală INSERTED poate avea mai multe linii (comanda a avut forma *INSERT INTO liniifact SELECT ... FROM ...*), de aceea primele comenzi UPDATE folosesc clauzele WHERE în forma prezentată, iar pentru numerotare este necesară folosirea unui cursor – *c_facturi* – care avea câte o înregistrare pentru fiecare factură inserată. Fiecărei facturi în care s-a adăugat măcar o linie i se (re)numerotează toate liniile, folosindu-se în acest scop un al doilea cursor – *c_linii*. Iată şi verificarea (vezi şi figura 17.12).



	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	3500	1	2	200	100.00	1800.00
2	3500	2	3	100	100.00	1900.00

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	3500	2008-12-17 10:22:00	1002	NULL	33700.00	3700.00	0.00	NULL	NULL

Figura 17.12. Verificarea funcţionării declanşatorului INSERT pentru tabela LINIIFACT

INSERT INTO liniifact VALUES (3500, 1, 2, 200, 100, 8989989);

INSERT INTO liniifact VALUES (3500, 4, 3, 100, 100, 0);

SELECT * FROM liniifact WHERE NrFact = 3500

SELECT * FROM facturi WHERE NrFact = 3500

Chiar dacă a doua comandă INSERT a încercat să atribuie celei de-a doua linii numărul 4, *trigger*-ul şi-a făcut datoria şi a asigurat numerotarea corectă.

Declanşatorul de ştergere de-acum este unul banal (vezi listingul 17.29). Trebuie recalculate valorile atributelor TVA şi ValTotală din tabela FACTURI şi renumerotate liniile în care s-a şters măcar o linie. Fireşte, de data aceasta primadona este tabela virtuală DELETED în care sunt înregistrate toate liniile şterse.

Listingul 17.29. Declanşatorul DELETE pentru tabela LINIIFACT

```
CREATE TRIGGER trg_liniifact_del ON liniifact FOR delete
AS
    UPDATE facturi
    SET TVA = (SELECT SUM(TVAlinie) FROM liniifact WHERE NrFact = facturi.NrFact),
        ValTotala = (SELECT SUM(Cantitate * PretUnit + TVAlinie)
                    FROM liniifact WHERE NrFact = facturi.NrFact)
    WHERE NrFact IN (SELECT NrFact FROM deleted)

    DECLARE @i TINYINT;
    DECLARE @linie TINYINT;
    DECLARE @nrfact INT;

    DECLARE c_facturi CURSOR FOR
        SELECT DISTINCT NrFact FROM deleted ORDER BY NrFact
    OPEN c_facturi
    FETCH NEXT FROM c_facturi INTO @nrfact
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @i = 1
        DECLARE c_linii CURSOR FOR
                SELECT Linie FROM liniifact WHERE NrFact =@nrfact ORDER BY Linie
        OPEN c_linii
        FETCH NEXT FROM c_linii INTO @linie
```

```
        WHILE @@FETCH_STATUS = 0 BEGIN
                UPDATE liniifact SET Linie = @i WHERE NrFact = @nrfact AND Linie = @linie
                SET @i = @i + 1
                FETCH NEXT FROM c_linii INTO @linie
        END
        CLOSE c_linii
        FETCH NEXT FROM c_facturi INTO @nrfact
    END
    DEALLOCATE c_linii
    CLOSE c_facturi
    DEALLOCATE c_facturi
```

Înainte de a verifica modul în care se comportă declanşatorul, îl redactăm şi pe cel de modificare (vezi listingul 17.30), folosind deopotrivă tabelele virtuale INSERTED şi DELETED. Avem şi două premiere. Mai întâi, vrem să interzicem modificarea directă (printr-o comandă UPDATE) a valorilor atributelor Linie şi TVALinie. Pentru aceasta ne vom folosi de funcţia *UPDATE()* care furnizează TRUE dacă valoarea atributului-argument a fost modificată şi FALSE în caz contrar, şi de funcţia *TRIGGER_NESTELEVEL()* care se incrementează cu valoarea 1 la fiecare declanşator lansat (care poate lansa, la rândul său, un alt declanşator şi, astfel, se mai creşte cu 1 valoarea furnizată de funcţie). Dacă modificarea s-ar face direct, atunci valoarea furnizată de funcţie ar fi 1.

Listingul 17.30. Declanşatorul UPDATE pentru tabela LINIIFACT

```
CREATE TRIGGER trg_liniifact_upd ON liniifact FOR update AS
IF (UPDATE(Linie) OR UPDATE(TVAlinie) ) AND (SELECT TRIGGER_NESTLEVEL() ) <= 1
    BEGIN
    RAISERROR('Nu aveti voie sa modificati interactiv valorile atributelor Linie si TVALinie',16, 1)
    ROLLBACK TRANSACTION
    END

IF NOT UPDATE(Linie)
    BEGIN
    PRINT TRIGGER_NESTLEVEL()
    UPDATE facturi
    SET TVA = (SELECT SUM(TVAlinie) FROM liniifact WHERE NrFact = facturi.NrFact),
        ValTotala = (       SELECT SUM(Cantitate * PretUnit + TVAlinie)
                            FROM liniifact WHERE NrFact = facturi.NrFact)
    WHERE NrFact IN (SELECT NrFact FROM deleted UNION
                                        SELECT NrFact FROM DELETED )

    DECLARE @i TINYINT;
    DECLARE @linie TINYINT;
    DECLARE @nrfact INT;
    DECLARE c_facturi CURSOR FOR
        SELECT DISTINCT NrFact FROM inserted UNION
                SELECT DISTINCT NrFact FROM deleted ORDER BY NrFact
    OPEN c_facturi
    FETCH NEXT FROM c_facturi INTO @nrfact
    WHILE @@FETCH_STATUS = 0 BEGIN
    PRINT @nrfact
        SET @i = 1
        DECLARE c_linii2 CURSOR FOR
                SELECT Linie FROM liniifact WHERE NrFact = @nrfact ORDER BY Linie
        OPEN c_linii2
        FETCH NEXT FROM c_linii2 INTO @linie
```

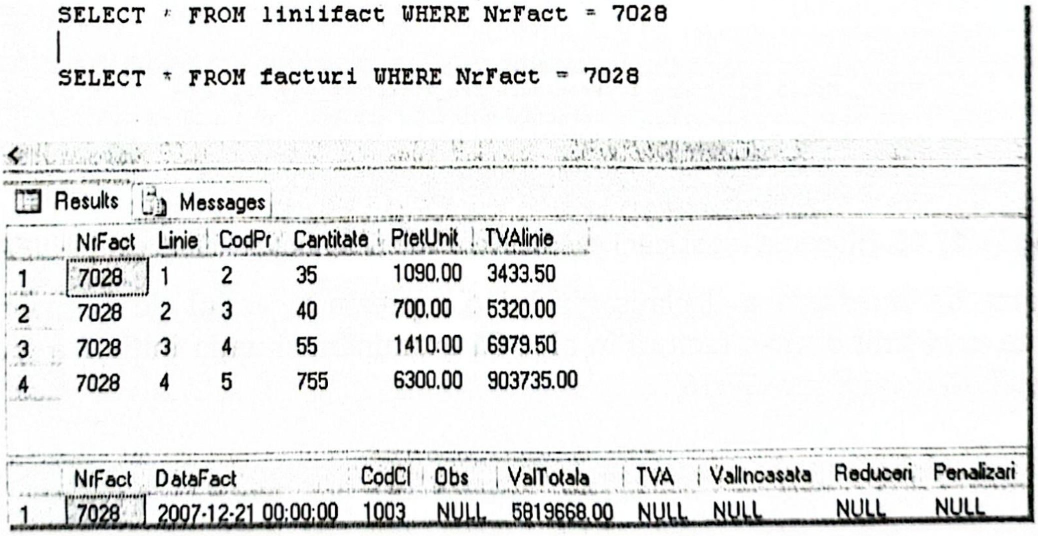
```
            WHILE @@FETCH_STATUS = 0
            BEGIN
                    PRINT @linie
                    UPDATE liniifact SET Linie = @i WHERE NrFact = @nrfact AND Linie = @linie
                    SET @i = @i + 1
                    FETCH NEXT FROM c_linii2 INTO @linie
            END
            CLOSE c_linii2
            DEALLOCATE c_linii2
            FETCH NEXT FROM c_facturi INTO @nrfact
        END
        CLOSE c_facturi
        DEALLOCATE c_facturi
    END
```

De asemenea, mai luăm în calcul faptul că renumerotarea liniilor, efectuată de acest declanşator sau de *trg_liniifact_del*, ar lansa, recursiv, *trg_liniifact_upd*. De aceea, toate prelucrările din acest declanşator nu vor fi efectuate dacă singura valoare (sau valori) modificată ar fi cea a atributului Linie. Cursorul *c_facturi* va fi populat cu numerele facturilor care au suferit modificări, numere obţinute prin reunirea valorilor NrFact din tabelele INSERTED şi DELETED.

Pentru o minitestare a declanşatoarelor, să ne concentrăm pe factura cu numărul 7028 (vezi figura 17.13):

SELECT * FROM liniifact WHERE NrFact = 7028

SELECT * FROM facturi WHERE NrFact = 7028



```
SELECT * FROM liniifact WHERE NrFact = 7028

SELECT * FROM facturi WHERE NrFact = 7028
```

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	7028	1	2	35	1090.00	3433.50
2	7028	2	3	40	700.00	5320.00
3	7028	3	4	55	1410.00	6979.50
4	7028	4	5	755	6300.00	903735.00

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	7028	2007-12-21 00:00:00	1003	NULL	5819668.00	NULL	NULL	NULL	NULL

Figura 17.13. Conţinutul „iniţial" al facturii 7028

Factura are patru linii în care apar produsele 2, 3, 4 şi 5. Le ştergem pe prima şi a treia:

DELETE FROM liniifact WHERE NrFact = 7028 AND Linie IN (1,3)

După cum o demonstrează factura 17.14, declanşatorul de ştergere recalculează valoarea totală şi TVA-ul (în tabela FACTURI) şi renumerotează corect cele două linii rămase.

```
SELECT * FROM liniifact WHERE NrFact = 7028
SELECT * FROM facturi WHERE NrFact = 7028

BEGIN TRAN
DELETE FROM liniifact WHERE NrFact = 7028 AND Linie IN (1,3)

SELECT * FROM liniifact WHERE NrFact = 7028
SELECT * FROM facturi WHERE NrFact = 7028
```

Results | Messages

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	7028	1	3	40	700.00	5320.00
2	7028	2	5	755	6300.00	903735.00

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	7028	2007-12-21 00:00:00	1003	NULL	5693555.00	909055.00	NULL	NULL	NULL

Figura 17.14. Conţinutul facturii 7028 după ştergerea liniilor 1 şi 3

Verificăm şi vigilenţa declanşatorului *trg_liniifact_upd* încercând să modificăm „direct" numărul primei linii din factura 7028:

UPDATE liniifact SET Linie = 6 WHERE NrFact = 7028 AND Linie = 1

Spre meritul lui, *trigger*-ul se opune acestei samavolnicii (vezi figura 17.15).

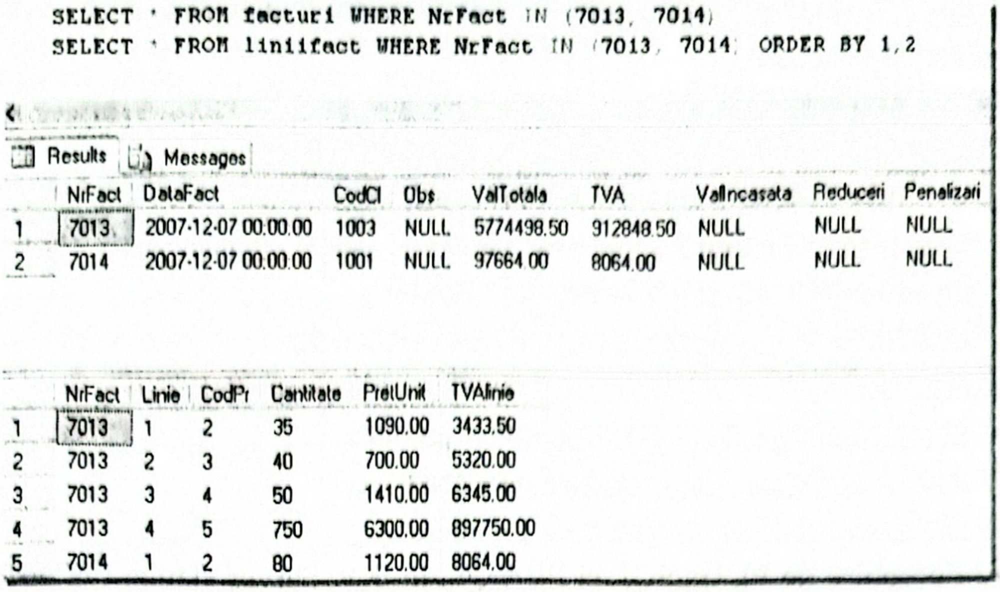
```
UPDATE liniifact SET Linie=7 WHERE NrFact = 7028 AND Linie = 1
```

Messages

```
Msg 50000, Level 16, State 1, Procedure trg_liniifact_upd, Line 6
Nu aveti voie sa modificati interactiv valorile atributelor Linie si TVALinie
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

Figura 17.15. Blocarea modificării interactive a valorii atributului Linie din LINIIFACT

Piatra de încercare a declanşatorului o reprezintă modul de comportare la trecerea unei linii dintr-o factură în alta. Să examinăm situaţia iniţială a facturilor 7013 şi 7014 (vezi figura 17.16).



```
SELECT * FROM facturi WHERE NrFact IN (7013, 7014)
SELECT * FROM liniifact WHERE NrFact IN (7013, 7014) ORDER BY 1,2
```

Results | Messages

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	7013	2007-12-07 00:00:00	1003	NULL	5774498.50	912848.50	NULL	NULL	NULL
2	7014	2007-12-07 00:00:00	1001	NULL	97664.00	8064.00	NULL	NULL	NULL

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	7013	1	2	35	1090.00	3433.50
2	7013	2	3	40	700.00	5320.00
3	7013	3	4	50	1410.00	6345.00
4	7013	4	5	750	6300.00	897750.00
5	7014	1	2	80	1120.00	8064.00

Figura 17.16. Situaţia de pornire a facturilor 7013 şi 7014

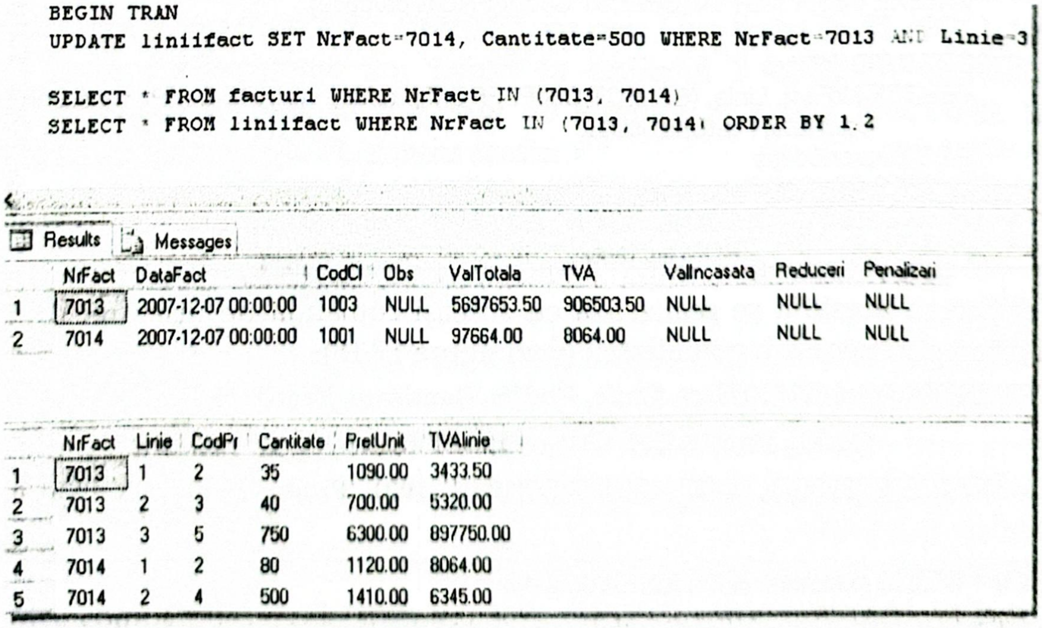
Prima are patru linii în care apar produsele 2, 3, 4 şi 5, iar a doua doar o linie ce indică vânzarea produsului 2. Încercăm să trecem a treia linie din factura 7013 în factura 7014. Comanda nu ar fi complicată:

UPDATE liniifact

SET NrFact = 7014, Cantitate = 500

WHERE NrFact = 7013 AND Linie = 3

Ceea ce aşteptăm noi de la declanşator este recalcularea valorii totale şi a TVA-ului pentru cele două facturi, dar şi renumerotarea liniilor. Figura 17.17 demonstrează că lucrurile au decurs conform planului.



```
BEGIN TRAN
UPDATE liniifact SET NrFact=7014, Cantitate=500 WHERE NrFact=7013 AND Linie=3

SELECT * FROM facturi WHERE NrFact IN (7013, 7014)
SELECT * FROM liniifact WHERE NrFact IN (7013, 7014) ORDER BY 1,2
```

Results | Messages

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	7013	2007-12-07 00:00:00	1003	NULL	5697653.50	906503.50	NULL	NULL	NULL
2	7014	2007-12-07 00:00:00	1001	NULL	97664.00	8064.00	NULL	NULL	NULL

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	7013	1	2	35	1090.00	3433.50
2	7013	2	3	40	700.00	5320.00
3	7013	3	5	750	6300.00	897750.00
4	7014	1	2	80	1120.00	8064.00
5	7014	2	4	500	1410.00	6345.00

Figura 17.17. Situaţia facturilor 7013 şi 7014 după mutarea unei linii

Încheiem cu un declanşator de tip INSTEAD OF. Folosim acelaşi exemplu cu tabela virtuală vFacturi:

```
CREATE VIEW vFacturi AS
SELECT lf.NrFact, DataFact, DenCl, f.CodCl, Linie, lf.CodPr, DenPr,
        UM, ProcTVA, Grupa, Cantitate, PretUnit,
        Cantitate * PretUnit AS ValFaraTVALinie,
        Cantitate * PretUnit * ProcTVA AS TVALinie,
        Cantitate * PretUnit * (1+ProcTVA) AS ValTotLinie
FROM liniifact lf
        INNER JOIN produse p ON lf.CodPr = p.CodPr
        INNER JOIN facturi f ON lf.NrFact = f.NrFact
        INNER JOIN clienti c ON f.CodCl = c.CodCl
```

Spre deosebire de PL/pgSQL şi PL/SQL, în SQL Server clauza ORDER BY nu poate apărea în interogarea-definiţie a tabelei virtuale. Declanşatorul INSTEAD OF din listingul 17.31 este croit după logica tabelei „logice" INSERT.

Listingul 17.31. Declanşatorul UPDATE pentru tabela LINIIFACT

```
CREATE TRIGGER trg_vFacturi_ioi ON vFacturi INSTEAD OF INSERT
AS
    INSERT INTO facturi (NrFact, DataFact, CodCl)
        SELECT NrFact, DataFact, CodCl
        FROM inserted
        WHERE NrFact NOT IN (SELECT NrFact FROM facturi)

    INSERT INTO produse (DenPr, UM, Grupa, ProcTVA)
        SELECT DenPr, UM, Grupa, ProcTVA
        FROM inserted
        WHERE CodPr NOT IN (SELECT CodPr FROM produse)

    INSERT INTO liniifact
        SELECT NrFact, Linie, (SELECT CodPr FROM produse WHERE DenPr = ins.DenPr),
                Cantitate, PretUnit, NULL
        FROM inserted ins
        WHERE CAST(NrFact AS CHAR(8)) + CAST (Linie AS CHAR(2))
                NOT IN (SELECT CAST(NrFact AS CHAR(8)) + CAST (Linie AS CHAR(2))
                        FROM liniifact)
```

Verificarea acestuia se realizează cu aceeaşi comandă de inserare, urmată de două interogări necesare vizualizării (vezi figura 17.18).

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
                CodCl, DenPr, UM, Grupa, ProcTVA)
        VALUES (3120, 1, 8, 400, 1200, '2007-10-01', 1001, 'Produs 8', 'buc', 'Cafea', 0.19);

SELECT * FROM produse WHERE CodPr = 8
SELECT * FROM liniifact WHERE NrFact = 3120
SELECT * FROM facturi WHERE NrFact = 3120
```

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
        CodCl, DenPr, UM, Grupa, ProcTVA)
   VALUES (3120, 1, 8, 400, 1200, '2007-10-01', 1001, 'Produs 8',
        'buc', 'Cafea', 0.19) ;

SELECT * FROM produse WHERE CodPr = 8
SELECT * FROM liniifact WHERE NrFact = 3120
SELECT * FROM facturi WHERE NrFact = 3120
```

Results | Messages

	CodPr	DenPr	UM	Grupa	ProcTVA
1	8	Produs 8	buc	Cafea	0.19

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVAlinie
1	3120	1	8	400	1200.00	91200.00

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata	Reduceri	Penalizari
1	3120	2007-10-01 00:00:00	1001	NULL	571200.00	91200.00	0.00	0.00	0.00

Figura 17.18. Verificarea declanşatorului INSTEAD OF

17.5. Declanşatoare în SQL PL

Tipologia declanşatoarelor este comparabilă, ca nivel de detaliere, cu cea din Oracle. Ca şi în T-SQL, trecem peste secvenţa şi declanşatorul necesare cheilor-surogat (cum a fost CodCl din CLIENŢI, pe care l-am tratat în PL/SQL), întrucât pentru toate cheile-surogat am folosit în capitolul 3 opţiunea GENERATED ALWAYS. Poposim, deci, la *trigger*-ul de inserare în tabela FACTURI. Iată în listingul 17.32 sintaxa SQL PL pentru acesta.

Listingul 17.32. Prima versiune a declanşatorului SQL PL de inserare în FACTURI

```
CREATE TRIGGER trg_facturi_ins
NO CASCADE BEFORE INSERT ON facturi
REFERENCING NEW AS n
FOR EACH ROW
MODE DB2SQL
vs: BEGIN ATOMIC
    SET n.TVA = 0;
    SET n.ValTotala = 0;
    SET n.ValIncasata = 0;
END vs
```

Pentru comparaţie cu celelalte paragrafe, folosim acelaşi INSERT:

```
INSERT INTO facturi VALUES (3500, CURRENT_DATE, 1002, NULL,
        123456, 234567, 345678, NULL, NULL)
```

Similar Oracle, suntem obligaţi să redactăm declanşatorul care să ţină loc de ON UPDATE CASCADE (vezi listingul 17.33).

Listingul 17.33. Declanşator SQL PL pentru substituirea opţiunii ON UPDATE CASCADE

```
CREATE TRIGGER trg_facturi_upd_a_r_nrfact
AFTER  UPDATE OF NRFACT ON facturi
REFERENCING  OLD AS OLD  NEW AS NEW  FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
UPDATE liniifact SET NrFact = NEW.NrFact WHERE NrFact = OLD.NrFact;
UPDATE incasfact SET NrFact = NEW.NrFact WHERE NrFact = OLD.NrFact;
END
```

Pentru declanşatorul dedicat inserării unei înregistrări în tabela LINIIFACT apar câteva modificări. Nu putem proceda ca în PL/SQL unde aveam un singur declanşator BEFORE-ROW, în care modificam valorile atributelor Linie şi TVALinie din LINIIFACT şi incrementam ValTotală şi TVA în FACTURI. SQL PL ne obligă să mutăm actualizarea atributelor calculate în declnşatorul AFTER ROW. Aşa că vom avea două declanşatoare pentru inserare *trg_liniifact_ins1* şi *trg_liniifact_ins2* (vezi listingul 17.34).

Listingul 17.34. Două declanşatoare SQL PL pentru inserarea unei înregistrări în LINIIFACT

```
-- declansatorul la nivel de linie de tipul BEFORE
CREATE TRIGGER trg_liniifact_ins1 NO CASCADE
BEFORE INSERT ON liniifact
REFERENCING  NEW AS NEW  FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
    DECLARE v_tvalinie DECIMAL(9,2);
    SET v_tvalinie = COALESCE(NEW.cantitate,0) * COALESCE(NEW.pretunit,0) *
        f_proctva(CAST (NEW.codpr AS SMALLINT));
    SET NEW.tvalinie = v_tvalinie;
    SET NEW.linie  = (SELECT COALESCE(MAX(Linie),0) + 1
                      FROM liniifact
                      WHERE NrFact = NEW.NrFact);
    END

-- declansatorul la nivel de linie de tipul AFTER
CREATE TRIGGER TRG_LINIIFACTURI_INS2
AFTER INSERT ON liniifact
REFERENCING NEW AS NEW  FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
   UPDATE facturi
   SET tva = COALESCE(tva,0) + NEW.tvalinie,
       valtotala = COALESCE(valtotala,0) + NEW.cantitate * NEW.pretunit + NEW.tvalinie
    WHERE NrFact = NEW.NrFact;
END
```

Testăm aceste două declanşatoare cu aceleaşi două comenzi INSERT din paragrafele anterioare :

INSERT INTO liniifact VALUES (3500, 1, 2, 200, 100, 8989989);

INSERT INTO liniifact VALUES (3500, 2, 3, 100, 100, 0);

„Portăm" acum declanşatorul de ştergere pentru LINIIFACT în varianta sa rigidă, de interzicere a ştergerii unei alte linii decât ultima pentru o factură (listingul 17.35). Declanşarea erorii se realizează prin comanda SIGNAL.

Listingul 17.35. Declanşatorul SQL PL pentru ştergerea unei înregistrări din LINIIFACT

```
CREATE TRIGGER trg_liniifact_del
AFTER DELETE ON liniifact
REFERENCING OLD AS OLD  FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
    DECLARE v_nrlinii SMALLINT;
    SET v_nrlinii = (SELECT MAX(linie) FROM liniifact WHERE nrfact = OLD.nrfact );
    IF (OLD.linie <> COALESCE(v_nrlinii,0)) THEN
        SIGNAL SQLSTATE '80000'
      SET MESSAGE_TEXT='Nu puteti sterge decat ultima linie dintr-o factura';
    END IF;
    UPDATE facturi
    SET tva = tva - OLD.tvalinie,
            valtotala = valtotala - OLD.cantitate * OLD.pretunit - OLD.tvalinie
    WHERE nrfact = OLD.nrfact;
END
```

Spre deosebire de Oracle, în DB2 la verificarea declanşatorului prin comanda:
DELETE FROM liniifact WHERE NrFact = 3500 AND linie = 1
nu se declanşează eroarea de mutanţă a tabelei, ci eroarea cauzată de ştergerea dintr-o factură a altei linii decât ultima (figura 17.19).

```
DELETE FROM liniifact WHERE NrFact=3500 AND linie=1

1 row(s) returned successfully.
------------------------------ Commands Entered ------------------------------
DELETE FROM liniifact WHERE NrFact=3500 AND linie=1;
------------------------------------------------------------------------------
DELETE FROM liniifact WHERE NrFact=3500 AND linie=1
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0438N  Application raised error with diagnostic text: "Nu puteti sterge
decit ultima linie dintr-o factura".  SQLSTATE=80000
```

Figura 17.19. Funcţionarea primei versiuni a declanşatorului de ştergere în LINIIFACT

Varianta după care în facturile în care s-a şters măcar o linie se numerotează automat liniile rămase este mult mai puţin traumatizantă în DB2 SQL PL, prin comparaţie cu Oracle PL/SQL. Nu avem nevoie de variabile publice, ci doar de două declanşatoare, unul la nivel de linie, *trg_liniifact_del*, care să decrementeze valoarea celor două atribute calculate din FACTURI şi unul la nivel de comandă, *trg_liniifact_del2*, care renumerotează liniile (vezi listingul 17.36, mai întâi ştergem *trg_liniifact_del* şi apoi lansăm scriptul).

Listingul 17.36. Două declanşatoare SQL PL de ştergere în LINIIFACT

```
CREATE TRIGGER trg_liniifact_del
AFTER DELETE ON liniifact
REFERENCING OLD AS OLD FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
    UPDATE facturi
    SET tva = tva - OLD.tvalinie,
            valtotala = valtotala - OLD.cantitate * OLD.pretunit - OLD.tvalinie
```

```
    WHERE nrfact = OLD.nrfact;
END

--------------------------------------------------------------------------------
CREATE TRIGGER trg_liniifact_del2
AFTER DELETE ON liniifact
REFERENCING OLD_TABLE AS oldtab
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    DECLARE i SMALLINT;
    FOR rec1 AS SELECT DISTINCT NrFact FROM oldtab DO
        SET i = 1;
        FOR rec_linii AS SELECT * FROM liniifact WHERE NrFact = rec1.NrFact ORDER BY Linie DO
            UPDATE liniifact SET linie = i
            WHERE NrFact = rec_linii.NrFact AND linie=rec_linii.linie;
            SET i = i + 1;
        END FOR;
    END FOR;
END
```

În schimb, protejarea atributelor calculate din tabela FACTURI de modificările neautorizate reprezintă o sarcina mult mai dificilă. Există în SQL PL variabile publice ce pot fi gestionate la nivel de sesiune de lucru, sintaxa fiind una simplă:

CREATE VARIABLE v_trg_liniifact CHAR(1) DEFAULT NULL

Din păcate, variabilele publice nu pot fi modificate în declanșatoare, așa că nu ne sunt de folos. Soluția tabelelor temporare (globale) reclamă cunoștințe de SQL dinamic și abia în capitolul următor vom încerca o soluție.

Așa că singura variantă ar fi să creăm o tabelă în care fiecărei variabile publice și utilizator să li se aloce câte o linie (înregistrare), urmând ca declanșatoarele tabelelor LINIIFACT și FACTURI să lucreze cu această tabelă-substitut a variabilelor publice.

Așa că încheiem paragraful cu exemplul de declanșatoare din categoria INSTEAD OF. Comanda CREATE VIEW nu acceptă în DB2 clauza ORDER BY, așa că sintaxa comenzii CREATE VIEW pentru vFacturi este:

```
CREATE VIEW vFacturi AS
        SELECT lf.NrFact, DataFact, DenCl, f.CodCl, Linie, lf.CodPr, DenPr,
                UM, ProcTVA, Grupa,
                Cantitate, PretUnit,
Cantitate * PretUnit AS ValFaraTVALinie,
                Cantitate * PretUnit * ProcTVA AS TVALinie,
                Cantitate * PretUnit * (1+ProcTVA) AS ValTotLinie
        FROM liniifact lf
                INNER JOIN produse p ON lf.CodPr = p.CodPr
                INNER JOIN facturi f ON lf.NrFact = f.NrFact
                INNER JOIN clienti c ON f.CodCl = c.CodCl
```

Față de versiunea PL/SQL, în afara diferențelor de sintaxă, în redactarea declanșatorului INSTEAD OF INSERT în DB2 SQL PL trebuie avut în vedere că atributul CodPr din tabela PRODUSE are valori generate automat, deci în afara

oricărui control. Ne bazăm pe faptul că denumirea produsului, deşi nu este cheie alternativă, nu se repetă în tabela PRODUSE (vezi listingul 17.37).

Listingul 17.37. Declanşatorul INSTEAD OF INSERT

```
CREATE TRIGGER trg_vFacturi_ioi
INSTEAD OF INSERT ON vFacturi
REFERENCING NEW AS NEW FOR EACH ROW  MODE DB2SQL
BEGIN ATOMIC
    INSERT INTO facturi (NrFact, DataFact, CodCl)
        SELECT NEW.NrFact, NEW.DataFact, NEW.CodCl
        FROM sysibm.dual
        WHERE NEW.NrFact NOT IN (SELECT NrFact FROM facturi);

    INSERT INTO produse (DenPr, UM, Grupa, ProcTVA)
        SELECT NEW.DenPr, NEW.UM, NEW.Grupa, NEW.ProcTVA
        FROM sysibm.dual
        WHERE NEW.CodPr NOT IN (SELECT CodPr FROM produse);

    INSERT INTO liniifact VALUES (NEW.NrFact, NEW.Linie,
            (SELECT CodPr FROM produse WHERE DenPr=NEW.DenPr) ,
            NEW.Cantitate,
        NEW.PretUnit, NULL);
END
```

Chiar dacă în INSERT-ul de mai jos codul produsului specificat este 8, în tabelele LINIIFACT şi PRODUSE a fost inserat codul 22 (vezi figura 17.20):

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
                CodCl, DenPr, UM, Grupa, ProcTVA)
        VALUES (3120, 1, 8, 400, 1200, '2007-10-01', 1001, 'Produs 8', 'buc', 'Cafea', 0.19);
SELECT * FROM produse WHERE CodPr = 8;
SELECT * FROM liniifact WHERE NrFact = 3120;
SELECT * FROM facturi WHERE NrFact = 3120;
```

```
SELECT * FROM liniifact WHERE NrFact = 3120

NRFACT      LINIE  CODPR    CANTITATE  PRETUNIT    TVALINIE
----------- ------ -------- ---------- ----------- -----------
      3120.      1      23.       400.     1200.00    91200.00

  1 record(s) selected.


SELECT * FROM produse

CODPR  DENPR                          UM         GRUPA           PROCTVA
------ ------------------------------ ---------- --------------- -------
     1 Produs 1                       buc        Tigari              .19
     2 Produs 2                       kg         Bere                .09
     3 Produs 3                       kg         Bere                .19
     4 Produs 4                       l          Dulciuri            .09
     5 Produs 5                       buc        Tigari              .19
     6 Produs 6                       p250g      Cafea               .19
    23 Produs 8                       buc        Cafea               .19

  7 record(s) selected.
```

Figura 17.20. Testarea declanşatorului INSTEAD OF vFacturi

Capitolul 18
SQL dinamic

Tema dedicată SQL-ului dinamic (Dynamic SQL) este vecină cu cea a SQL-ului împachetat (Embedded SQL). Cea din urmă privește modul în care comenzile SQL pot fi incluse în programe scrise în Java, C, Basic, Cobol etc. Noi însă ne vom ocupa de prima care privește, în principal, modalitățile în care programele (blocurile) PL/pgSQL, PL/SQL, T-SQL și SQL PL conțin comenzi DDL și DML în care numele obiectelor (tabele, view-uri, atribute, proceduri etc.) sunt cunoscute abia în momentul execuției, nu la scrierea (compilarea) programului.

Cum deja v-ați obișnuit, nu voi epuiza subiectul, ci mă voi limita doar la câteva aspecte și avantaje, trecând sub tăcere dezavantajele/pericolele SQL-ului dinamic, cum ar fi viteza redusă de execuție și injecțiile SQL (*SQL injections*).

18.1. SQL dinamic în PostgreSQL PL/pgSQL

PostgreSQL este destul de tolerant în materie de creare de tabele prin programe. Astfel, funcția-procedură *p_test* (listingul 18.1) conține o comandă CREATE TABLE. Blocul funcționează fără probleme la apelare:

SELECT p_test()

Listingul 18.1. Funcție simplă PL/pgSQL ce creează o tabelă (la fel de simplă)

```
CREATE OR REPLACE FUNCTION p_test() RETURNS VOID
AS $$
BEGIN
    CREATE TABLE test (x NUMERIC(5), y VARCHAR(50));
END;
$$ LANGUAGE plpgsql;
```

Al doilea exemplu prefațează problema arhivării facturilor, în jurul căreia vor gravita exemplele din acest capitol. Pentru a păstra tabela FACTURI cât mai suplă, ne propunem să creăm copii „fragmentate" ale acesteia, cu numele FACTURI_2007_8, FACTURI_2007_9... . Numele tabelelor-arhivă indică anul și luna în care au fost emise facturile respective. Până să le creăm, redactăm o funcție care să șteargă toate eventualele arhive (vezi listingul 18.2).

Listingul 18.2. Funcție-procedură PL/pgSQL ce șterge toate „arhivele" tabelei FACTURI

```
CREATE OR REPLACE FUNCTION p_sterge_facturi1()
    RETURNS void AS
$$
DECLARE
    rec_nume VARCHAR(30);
BEGIN
```

```
        FOR rec_nume IN (SELECT tablename FROM pg_tables
                WHERE schemaname = 'public' AND tablename LIKE 'facturi_%') LOOP
            EXECUTE 'DROP TABLE ' || rec_nume ;
        END LOOP;
    END;
    $$ LANGUAGE 'plpgsql'
```

Funcția-procedură șterge toate tabelele ale căror nume încep cu *FACTURI_*. Printr-un concurs de împrejurări, existau două arhive anuale, FACTURI_2006 și FACTURI_2007. Prin lansarea procedurii:

SELECT p_sterge_facturi1()

scăpăm de ele. Descoperim, însă, și un mod elegant de a bloca PostgreSQL-ul. Dacă după lansare uităm să reîmprospătăm meniul arborescent (cu ajutorul butonului *Refresh* sau al opțiunii cu același nume din meniul contextual) și „scăpăm" un click pe numele uneia dintre cele două tabele proaspăt șterse, avem șanse mari să provocăm înghețarea PostgreSQL-ului (vezi figura 18.1).

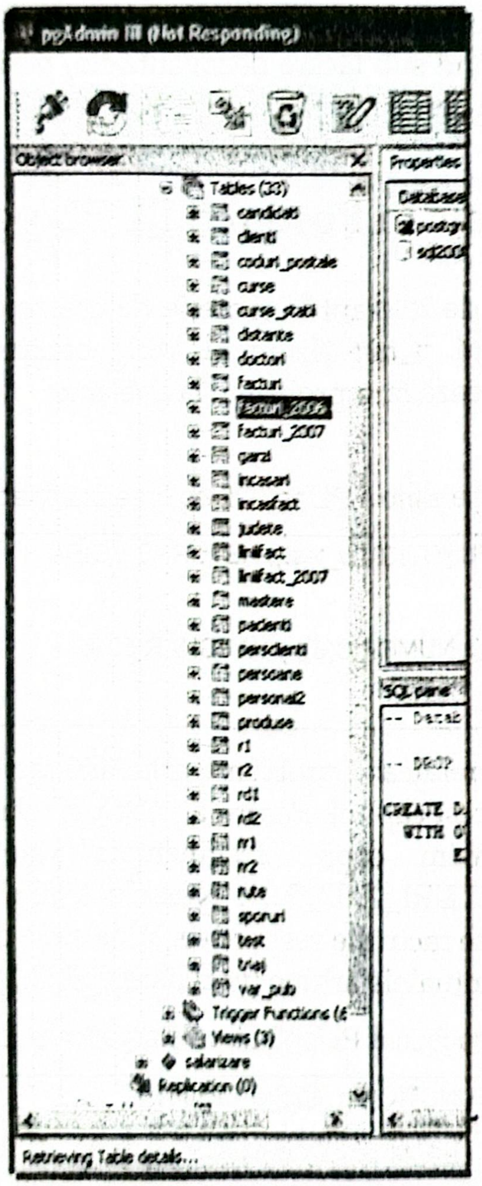


Figura 18.1. O sugestie (dezinteresată) pentru blocarea PostgreSQL-ului

Trecem la probleme ceva mai serioase. Acum, dacă tot merge strună funcția-procedură de ștergere, ar trebui să redactăm și procedura care să creeze arhivele. De data aceasta vom lucra mai îngrijit, adică mai parametrizat. Mai întâi, vom redacta o procedură de ștergere care va elimina din bază o arhivă a tabelei FACTURI pentru un an și o lună date. Eliminarea arhivelor este însă mai bine pusă la punct (prin comparație cu funcția din listingul 18.2), pentru că datele legate de facturi se găsesc și în tabelele LINIIFACT, ÎNCASĂRI (e un pic forțată includerea acestei tabele) și ÎNCASFACT. Așadar, se vor șterge toate cele patru eventuale arhive (vezi listingul 18.3).

Listingul 18.3. Funcție-procedură PL/pgSQL ce șterge toate „arhivele" tabelei FACTURI

```
CREATE OR REPLACE FUNCTION p_sterge_arhive_fact_luna
        (an_ SMALLINT, luna_ SMALLINT) RETURNS void AS
$$
DECLARE
    rec_nume VARCHAR(30);
    v_sir VARCHAR(15);
BEGIN
    v_sir := '_' || an_ || '_' || luna_;
    FOR rec_nume IN (SELECT tablename FROM pg_tables WHERE schemaname = 'public'
                AND tablename IN ( 'incasfact' || v_sir, 'liniifact' || v_sir, 'incasari' || v_sir,
                        'facturi' || v_sir) ) LOOP
        EXECUTE 'DROP TABLE ' || rec_nume ;
    END LOOP;
END;
$$ LANGUAGE 'plpgsql'
```

Ordinea ștergerii nu este importantă, întrucât între cele patru tabele nu sunt definite restricții referențiale, după cum vom vedea în continuare. Procedura--funcție din listingul 18.4 creează, pentru un an și o lună (*an_*, *luna_*) date, tabelele: FACTURI*_an_luna_*, LINIIFACT*_an_luna_*, ÎNCASĂRI*_an_luna_* și ÎNCAS-FACT*_an_luna_*. Crearea este precedată de ștergerea celor patru tabele-arhivă, folosind procedura anterioară. Se evită, astfel, eroarea declanșată la tentativa de creare a unei tabele cu un nume existent în schema curentă.

Listingul 18.4. Funcție-procedură PL/pgSQL ce șterge toate „arhivele" tabelei FACTURI

```
CREATE OR REPLACE FUNCTION p_creare_arhive_fact_luna(an_ SMALLINT, luna_ SMALLINT)
    RETURNS void AS
$$
DECLARE
    v_sir VARCHAR(15);
BEGIN
    EXECUTE p_sterge_arhive_fact_luna(an_, luna_);
    v_sir := '_' || an_ || '_' || luna_;
    EXECUTE 'CREATE TABLE facturi' || v_sir || ' AS SELECT * FROM facturi
        WHERE  EXTRACT (YEAR FROM DataFact) = '|| an_ ||
            ' AND EXTRACT(MONTH FROM DataFact) = ' || luna_;
    EXECUTE 'CREATE TABLE liniifact' || v_sir || ' AS SELECT * FROM liniifact
        WHERE NrFact IN (SELECT NrFact FROM facturi' || v_sir || ' ) ';
    EXECUTE 'CREATE TABLE incasari' || v_sir || ' AS SELECT * FROM incasari WHERE
        EXTRACT (YEAR FROM DataInc) = ' || an_ ||
            ' AND EXTRACT(MONTH FROM DataInc) = ' || luna_ ||
        ' UNION SELECT * FROM incasari WHERE CodInc IN (
```

```
                    SELECT CodInc FROM incasfact WHERE NrFact IN (
                              SELECT NrFact FROM facturi WHERE
                    EXTRACT (YEAR FROM DataFact)=' || an_ ||
                    ' AND EXTRACT(MONTH FROM DataFact) = ' || luna_ || ' )) ';
        EXECUTE 'CREATE TABLE incasfact' || v_sir || ' AS SELECT * FROM incasfact
            WHERE CodInc IN (SELECT CodInc FROM incasari' || v_sir || ') ';
END;
$$ LANGUAGE 'plpgsql'
```

În mod normal, ar trebui ca, după arhivare, să ştergem înregistrările plasate în tabele-arhivă din cele patru tabele „principale" - FACTURI, LINIIFACT, ÎNCASĂRI şi ÎNCASFACT. Deocamdată, nu avem curajul acesta, câştigul din folosirea tabelelor-arhivă fiind unul limitat. Testăm funcţionarea procedurii-funcţie:

SELECT p_creare_arhive_fact_luna(CAST (2007 AS SMALLINT),

CAST (1 AS SMALLINT))

Figura 18.2 surprinde ceea ce afişează meniul arborescent PostgreSQL după lansarea funcţiei (şi reîmprospătarea meniului). Faptul că cele patru tabele-arhivă sunt cam goale în cazul nostru are o importanţă secundară.

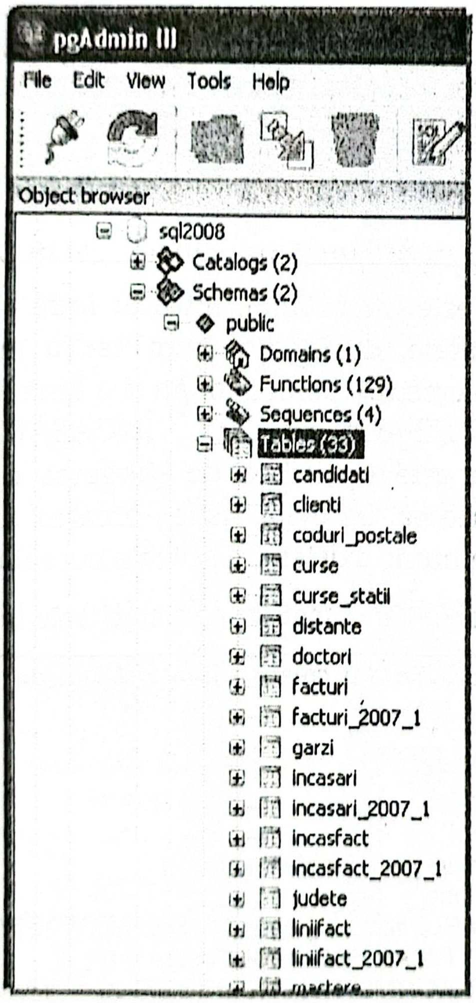


Figura 18.2. Verificarea procedurii de arhivare a facturilor (pentru anul 2007, luna 1)

Mai creăm tabele-arhivă şi pentru lunile august, septembrie şi octombrie 2007:

SELECT p_creare_arhive_fact_luna(CAST (2007 AS SMALLINT),

```
        CAST (8 AS SMALLINT));
SELECT p_creare_arhive_fact_luna(CAST (2007 AS SMALLINT),
        CAST (9 AS SMALLINT));
SELECT p_creare_arhive_fact_luna(CAST (2007 AS SMALLINT),
    CAST (10 AS SMALLINT));
```

Înainte de a redacta cea mai interesantă funcţie, cea care obţine setul de facturi pentru orice interval calendaristic pe baza tabelelor-arhivă, revenim la o problemă legată de declanşatoare. În paragraful 17.2 formulăm problema protejării atributelor calculate ale tabelei FACTURI de modificarea lor directă şi acceptarea doar a modificărilor produse de declanşatoarele tabelei LINIIFACT. Neavând la dispoziţie variabile publice (precum cele din Oracle), am creat o tabelă în care adăugăm, pentru fiecare utilizator conectat la baza de date, câte o înregistrare pentru fiecare variabilă publică (cum ar fi *v_trg_liniifact*).

Soluţia are câteva limite. Poate cea mai importantă este legată de faptul că pot exista două conexiuni la baza de date ale aceluiaşi utilizator. Este de evitat ca persoane diferite să folosească acelaşi cont la folosirea bazei de date. Se mai întâmplă, însă. Plus că, uneori, pentru a face diferite teste, un utilizator poate să deschidă două sau mai multe sesiuni pe acelaşi calculator. Dacă mai multe sesiuni operează editări ale tabelelor FACTURI şi LINIIFACT, declanşatoarele din paragraful 17.2 nu sunt în stare să recunoască decât un singur utilizator, iar consecinţele pot fi neplăcute.

Pentru a elimina acest neajuns, ne putem gândi la folosirea unei tabele temporare în locul celei „clasice". În PostgreSQL, tabelele temporare sunt locale, deci se vor şterge la finalizarea sesiunii. Este exact ce ne trebuie pentru a gestiona modificarea conţinului tabelelor FACTURI şi LINIIFACT. Funcţia *f_variabile_publice* din listingul 18.5 verifică existenţa tabelei temporare VARIABILE_PUBLICE şi, dacă este nevoie, o creează. De asemenea, se verifică dacă în tabela temporară se găseşte o înregistrare pentru variabila publică parametru (*variabila_*). În caz că se găseşte, se returnează valoarea curentă a acesteia.

Listingul 18.5. Funcţie-procedură ce creează o tabelă temporară locală pe post de variabilă publică

```
-- functia F_VARIABILE_PUBLICE creeaza, la prima apelare, tabela VARIABILE_PUBLICE
-- si, daca e nevoie, insereaza cate o linie pentru fiecare variabila publica: V_TRG_LINIIFACT...
-- pseudovariabilele de tip BOOLEAN sunt pe setate, la creare, pe FALSE
CREATE OR REPLACE FUNCTION f_variabile_publice (variabila_ VARCHAR)
        RETURNS BOOLEAN AS $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_tables WHERE tablename = 'variabile_publice') THEN
        EXECUTE 'CREATE LOCAL TEMPORARY TABLE variabile_publice (
                variabila VARCHAR(50),
                valoare BOOLEAN) ';
        INSERT INTO variabile_publice VALUES (variabila_, FALSE);
        RETURN FALSE;
    ELSE
        IF NOT EXISTS (SELECT 1 FROM variabile_publice WHERE variabila = variabila_) THEN
                INSERT INTO variabile_publice VALUES (variabila_, FALSE);
                RETURN FALSE;
        ELSE
```

```
                RETURN (SELECT valoare FROM variabile_publice WHERE variabila = variabila );
        END IF;
    END IF;
END;
$$LANGUAGE plpgsql ;
```

Acum trebuie să refacem şi declanşatoarele. Să începem cu cel de modificare pentru FACTURI (vezi listingul 18.6).

Listingul 18.6. Noua variantă a declanşatorului de modificare pentru FACTURI

```
-- se verifică modul de modificare a atributelor TVA si VALTOTALA
CREATE OR REPLACE FUNCTION trg_facturi_upd()
    RETURNS TRIGGER AS $trg_facturi_upd$
BEGIN
    IF NEW.tva <> OLD.tva OR NEW.valtotala <> OLD.valtotala THEN
        IF f_variabile_publice ('v_trg_liniifact') = FALSE THEN
                RAISE EXCEPTION
                'Nu puteti modifica interactiv nici valoarea facturii, nici TVA-ul acesteia!';
        END IF;
    END IF;
    RETURN NEW;
END;
$trg_facturi_upd$ LANGUAGE plpgsql;
```

```
DROP TRIGGER IF EXISTS trg_facturi ON facturi;
DROP TRIGGER IF EXISTS trg_facturi_upd ON facturi;
CREATE TRIGGER trg_facturi_upd AFTER UPDATE ON facturi
        FOR EACH ROW EXECUTE PROCEDURE trg_facturi_upd();
```

Continuăm cu declanşatoarele tabelei LINIIFACT: cel de inserare (vezi listingul 18.7), cel de modificare (listingul 18.8) şi cel de ştergere (listingul 18.9).

Listingul 18.7. Noua variantă a declanşatorului de inserare pentru LINIIFACT

```
CREATE OR REPLACE FUNCTION trg_liniifact_ins() RETURNS TRIGGER AS $trg_liniifact_ins$
DECLARE
    v BOOLEAN;
BEGIN
    v := p_variabile_publice('liniifact');
    UPDATE variabile_publice SET valoare = TRUE WHERE variabila = 'v_trg_liniifact';

    NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);
    UPDATE facturi SET tva = tva + NEW.tvalinie,
                valtotala = valtotala + NEW.cantitate * NEW.pretunit + NEW.tvalinie
        WHERE nrfact = NEW.nrfact;

    -- refacem valoarea pseudovariabilei
    UPDATE variabile_publice SET valoare = FALSE WHERE variabila = 'v_trg_liniifact';
    RETURN NEW;
END;
$trg_liniifact_ins$ LANGUAGE plpgsql;
```

```
DROP TRIGGER IF EXISTS trg_liniifact_ins ON liniifact;
CREATE TRIGGER trg_liniifact_ins BEFORE INSERT ON liniifact
        FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_ins();
```

Listingul 18.8. Noul declanşator de modificare pentru LINIIFACT

```
CREATE OR REPLACE FUNCTION trg_liniifact_upd_br()
        RETURNS TRIGGER AS $trg_liniifact_upd_br$
DECLARE
    v BOOLEAN;
BEGIN
    v := f_variabile_publice('v_trg_liniifact');
    UPDATE variabile_publice SET valoare = TRUE WHERE variabila = 'v_trg_liniifact';

    IF NEW.nrfact = OLD.nrfact THEN          -- nu s-a schimbat nrfact!!!!
        IF NEW.codpr <> OLD.codpr OR NEW.cantitate <> OLD.cantitate OR
                    NEW.pretunit <> OLD.pretunit OR NEW.tvalinie <> OLD.tvalinie THEN
            NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);
            UPDATE facturi SET tva = tva - OLD.tvalinie + NEW.tvalinie,
                    valtotala = valtotala - (OLD.cantitate * OLD.pretunit + OLD.tvalinie)
                                + (NEW.cantitate * NEW.pretunit + NEW.tvalinie)
            WHERE nrfact = NEW.nrfact;
        END IF;
    ELSE -- vechile valori trebuie scazute de la "vechea" factura
    -- iar noile valori adaugate "noii" facturi
        UPDATE facturi SET tva = tva - OLD.tvalinie,
                    valtotala = valtotala - (OLD.cantitate * OLD.pretunit + OLD.tvalinie)
        WHERE nrfact = OLD.nrfact;

        NEW.tvalinie := NEW.cantitate * NEW.pretunit * f_proctva(NEW.codpr);
        UPDATE facturi SET tva = tva + NEW.tvalinie,
                    valtotala = valtotala + (NEW.cantitate * NEW.pretunit + NEW.tvalinie)
        WHERE nrfact = NEW.nrfact;
    END IF;
    -- refacem valoarea pseudovariabilei
    UPDATE variabile_publice SET valoare = FALSE WHERE variabila = 'v_trg_liniifact';
    RETURN NEW;
END;
$trg_liniifact_upd_br$ LANGUAGE plpgsql;
```

DROP TRIGGER IF EXISTS trg_liniifact_upd_br ON liniifact;
CREATE TRIGGER trg_liniifact_upd_br BEFORE INSERT ON liniifact
FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_upd_br();

Listingul 18.9. Declanşatorul de ştergere pentru tabela LINIIFACT

```
CREATE OR REPLACE FUNCTION trg_liniifact_del2()
    RETURNS TRIGGER AS $trg_liniifact_del2$
DECLARE
    v BOOLEAN;
BEGIN
    v := f_variabile_publice('v_trg_liniifact');
    UPDATE variabile_publice SET valoare = TRUE WHERE variabila = 'v_trg_liniifact';
    UPDATE facturi
        SET tva = tva - OLD.tvalinie,
        valtotala = valtotala - OLD.cantitate * OLD.pretunit - OLD.tvalinie
        WHERE nrfact = OLD.nrfact;
    UPDATE variabile_publice SET valoare = FALSE WHERE variabila = 'v_trg_liniifact';
    RETURN OLD;
END;
$trg_liniifact_del2$ LANGUAGE plpgsql;
```

DROP TRIGGER IF EXISTS trg_liniifact_del2 ON liniifact;
CREATE TRIGGER trg_liniifact_del2 BEFORE DELETE ON liniifact

FOR EACH ROW EXECUTE PROCEDURE trg_liniifact_del2();

Ajungem, astfel, şi la funcţia care fuzionează arhivele pentru a furniza setul de facturi pentru orice interval dorit de către utilizator. Pentru încălzire, redactăm o funcţie care verifică existenţa unei tabele-parametru în schema PUBLIC (vezi listingul 18.10).

Listingul 18.10. Funcţie ce semnalizează existenţa (sau inexistenţa) unei tabele

```
CREATE OR REPLACE FUNCTION f_exista_tabela (nume_ VARCHAR)
    RETURNS BOOLEAN AS
$$
BEGIN
    RETURN EXISTS (SELECT tablename FROM pg_tables WHERE schemaname = 'public'
                AND tablename = nume_);
END;
$$ LANGUAGE 'plpgsql'
```

Listingul 18.11 conţine funcţia în care punem în valoare adevărata forţă a SQL-ului dinamic în PL/pgSQL. Bucla principală se execută pentru fiecare an din intervalul *data_inceput* – *data_final*. Bucla inclusă în aceasta parcurge toate lunile de pe anul în curs (*v_an*). Pentru primul an din interval, se începe cu luna din *data_inceput* şi se termină cu 12 (luna decembrie). În cazul anului final din interval, luna de început este 1 (ianuarie), iar cea finală este extrasă din *data_final*. Pentru lunile de început şi sfârşit se face o filtrare suplimentară, prin variabila *clauza_where*. Pentru fiecare combinaţie (*an_*, *luna_*), se verifică dacă există tabela-arhivă corespunzătoare şi, dacă da, se foloseşte SQL-ul dinamic şi o variabilă cursor care va returna toate înregistrările (eventual, filtrate) din tabela-arhivă pentru luna/anul curente care nu se găsesc în tabelele principale.

Listingul 18.11. Funcţie de reconstituire a facturilor pentru un interval calendaristic oarecare

```
CREATE OR REPLACE FUNCTION f_facturi (data_inceput DATE, data_final date)
    RETURNS SETOF facturi AS
$$
DECLARE
    v_an SMALLINT;
    v_luna SMALLINT;
    v_luna_inceput SMALLINT;
    v_luna_final SMALLINT;
    v_nume VARCHAR(30);
    rec1 RECORD;
    v_clauza_where VARCHAR(1000) := ' ';
BEGIN
    FOR v_an IN EXTRACT(YEAR FROM data_inceput)..EXTRACT(YEAR FROM data_final)
        LOOP

        v_luna_inceput = CASE WHEN v_an = EXTRACT(YEAR FROM data_inceput)
                    THEN EXTRACT(MONTH FROM data_inceput)
                    ELSE 1
            END;
        v_luna_final = CASE WHEN v_an = EXTRACT(YEAR FROM data_final)
                    THEN EXTRACT(MONTH FROM data_final)
                    ELSE 12
            END;
```

```
            FOR v_luna IN v_luna_inceput..v_luna_final LOOP
                v_nume := 'facturi_' || v_an || '_' || v_luna;

                IF f_exista_tabela(v_nume) THEN
                        v_clauza_where := ' WHERE NrFact NOT IN (SELECT NrFact FROM facturi) ' ;
                        IF v_an = EXTRACT(YEAR FROM data_inceput) AND v_luna =
                                        EXTRACT(MONTH FROM data_inceput) THEN
                                v_clauza_where := v_clauza_where ||
                                        ' AND DataFact >= DATE''' || data_inceput || '''' ;
                        END IF;
                        IF v_an = EXTRACT(YEAR FROM data_final) AND v_luna =
                                        EXTRACT(MONTH FROM data_final) THEN
                                v_clauza_where := v_clauza_where ||
                                        ' AND DataFact <= DATE''' || data_final || '''';
                        END IF;
                        FOR rec1 IN EXECUTE 'SELECT * FROM ' || v_nume ||
                                        v_clauza_where LOOP
                                RETURN NEXT rec1;
                        END LOOP;
                END IF;

            END LOOP;
        END LOOP;
        FOR rec1 IN EXECUTE ' SELECT * FROM facturi WHERE DataFact BETWEEN DATE''' ||
                data_inceput || '''' || ' AND DATE''' || data_final || '''' LOOP
            RETURN NEXT rec1;
        END LOOP;
END;
$$ LANGUAGE 'plpgsql'
```

După ieşirea din bucla principală, se parcurg şi liniile „rămase" în tabela FACTURI, care se încadrează în intervalul dat. Această operaţiune este necesară deoarece, la momentul execuţiei funcţiei *f_facturi*, este posibil ca arhivarea să se fi făcut numai pentru o parte dintre lunile intervalului sau poate chiar pentru niciuna. La testarea funcţiei descoperim că lucrurile chiar sunt în regulă (vezi figura 18.3):

SELECT f.nrfact, f.datafact, f.valtotala

FROM f_facturi (DATE'2007-08-04', DATE'2007-08-09') f

ORDER BY 1

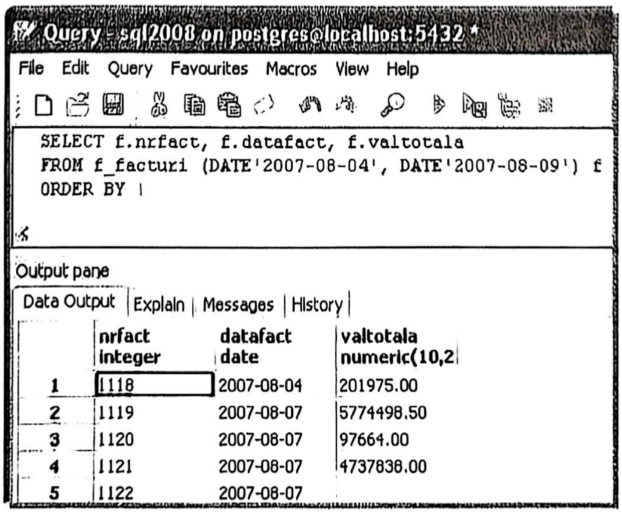


Figura 18.3. Verificarea procedurii de reconstituire a facturilor

18.2. SQL dinamic în PL/SQL

Oracle PL/SQL pune la dispoziţie două tipuri de SQL dinamic, prin pachetul standard DBMS_SQL (furnizat la instalare) şi prin comanda EXECUTE IMMEDIATE. Noi vom folosi exclusiv a doua variantă, referită în documentaţii drept NDS (Native Dynamic SQL). Din capul locului trebuie precizat că orice modul PL/SQL este necooperant în materie de comenzi DDL (vezi, pentru ilustrare, blocul anonim din figura 18.4).

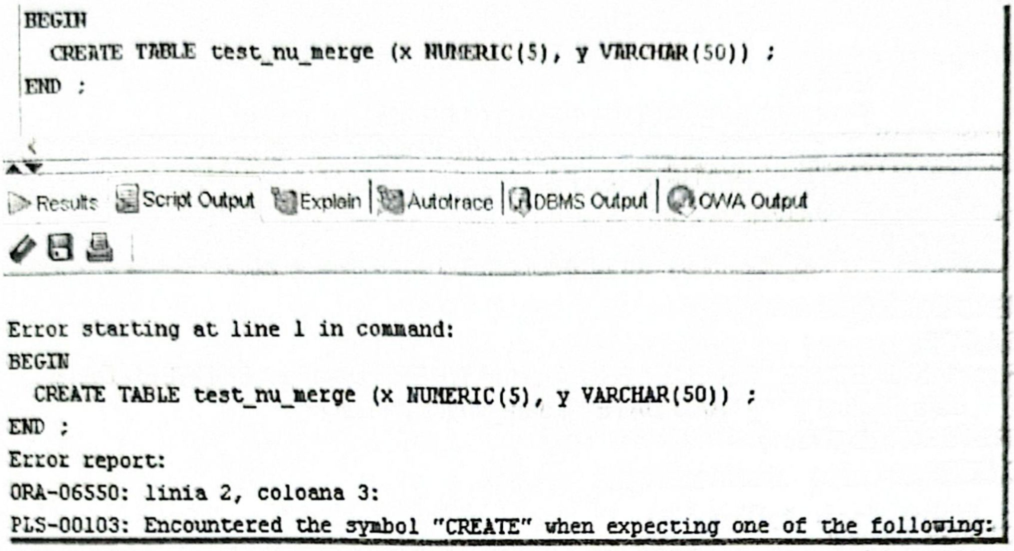


Figura 18.4. Imposibilitatea lansării unei comenzi CREATE TABLE dintr-un bloc PL/SQL

În paragraful 16.3, spuneam că puteam salva un set de înregistrări într-o tabelă temporară şi, totodată, amânăm discuţia deocamdată. Beneficiem de avantajul pachetelor PL/SQL, aşa că grupăm toate funcţiile şi procedurile de arhivare în *pac_arhivare*. După cum se poate vedea în listingul 18.12, pachetul conţine două „instanţe" ale funcţiei *f_există_tabelă*, o procedură care şterge arhivele pentru o lună şi un an date, *p_şterge_arhive_fact*, o procedură care face arhivarea pentru un an/lună date, *p_creare_arhive_fact_luna*, o funcţie care, pentru intervalul (*data_început*, *data_final*) dat, furnizează şirul de caractere (SELECT * ... UNION SELECT *...) care, lansat în execuţie, va reconstitui toate facturile emise în perioada respectivă.

Listingul 18.12. Specificaţiile pachetului *pac_arhivare*

```
==============================================================
-- pachetul pac_arhivare este cel care exemplifica o serie de optiuni de tip SQL dinamic
==============================================================
CREATE OR REPLACE PACKAGE pac_arhivare AUTHID CURRENT_USER AS

-- o functie "supraincarcata" ce verifica existenta unei tabele
FUNCTION f_exista_tabela (tabela_ VARCHAR2, an_ NUMBER, luna_ NUMBER)
    RETURN BOOLEAN;
FUNCTION f_exista_tabela (tabela_ VARCHAR2) RETURN BOOLEAN;

-- procedura ce sterge o arhiva lunara
PROCEDURE p_sterge_arhive_fact_luna (an_ NUMBER, luna_ NUMBER);
```

```
-- procedura P_CREARE_ARHIVE_FACT_LUNA creeaza o arhiva pentru fiecare tabela
"tranzactionala"
-- cu inregistrarile pe un an si luna, apoi sterge din tabela principala inregistrarile
-- arhivate (de exemplu, FACTURI pentru luna 8 din 2007 – se creeaza FACTURI_2007_8
-- se muta acolo toate facturile emise (DataFact) in luna august 2007
PROCEDURE p_creare_arhive_fact_luna (an_ NUMBER, luna_ NUMBER);

------------------------------------------------------------------------------------------
-- urmeaza o procedura si o functie pentru "recompunerea" dinamica a inregistrarilor arhivate

-- sirul care reuneste toate facturile pentru un interval dat
FUNCTION f_sir_facturi (data_inceput DATE, data_final DATE) RETURN VARCHAR2;

-- varianta 1 de reconstituire facturi : tabela temporara globala
PROCEDURE p_fuziune_facturi1 (datai DATE, dataf DATE) ;

END pac_arhivare;
--=====================================================================
```

Există câteva comentarii importante şi despre corpul pachetului (listingul 18.13). Procedura de ştergere a arhivelor (*p_şterge_arhive_fact_luna*) este ceva mai precaută. Pentru a nu se pierde iremediabil înregistrările care se găsesc în tabelele--arhivă, acestea (înregistrările) se readaugă în tabelele din care au fost extrase la momentul arhivării. Adăugarea trebuie făcută selectiv pentru a nu se încălca restricţiile de cheie primară. În plus, pentru ca *trigger*-ul *trg_liniifact_ins* să nu genereze eroarea de mutanţă a tabelei LINIIFACT, în loc de *INSERT INTO liniifact SELECT...*, folosim o variabilă cursor pentru a reintroduce eventualele înregistrări din arhivă în LINIIFACT una câte una.

Întrucât înregistrările din arhive trebuie adăugate în tabelele principale fără a se mai executa declanşatoarele, am inclus în procedura de arhivare şi comenzi EXECUTE IMMEDIATE de dezactivare şi, ulterior, reactivare a declanşatoarelor *trg_facturi_ins* şi *trg_liniifact_ins*. Exemplul este mai mult didactic, întrucât activarea şi dezactivarea declanşatoarelor este nerecomandată în aplicaţii complexe, cu mulţi utilizatori. Mai nimerită era modificarea declaşatoarelor pentru a nu executa anumite comenzi în cazul arhivării/dezarhivării (putem folosi una sau mai multe variabile publice în acest scop). Observaţia este valabilă şi pentru dezactivarea/activarea declanşatoarelor de ştergere din procedura de arhivare.

Arhivarea facturilor şi a încasărilor pentru un an şi lună date (*p_creare_arhive_fact_luna*) poate fi făcută acum fără grijă şi începe cu apelul procedurii de ştergere despre care vorbeam acum câteva rânduri. De notat şi, eventual, corectat (dacă nu sunt în regulă) predicatele aplicate la arhivare celor patru tabele tranzacţionale.

Listingul 18.13. Corpul pachetului *pac_arhivare*

```
--=====================================================================
    CREATE OR REPLACE PACKAGE BODY "PAC_ARHIVARE" AS
--=====================================================================
------------------------------------------------------------------------
FUNCTION f_exista_tabela(tabela_ VARCHAR2, an_ NUMBER, luna_ NUMBER)
  RETURN boolean
IS
  v_tabela VARCHAR2(100);
  v_unu NUMBER(1);
```

```
BEGIN
  v_tabela := UPPER(tabela_) || '_' || LTRIM(to_char(an_,   '9999')) || '_' || LTRIM(to_char(luna_,   '99'));
  SELECT 1 INTO v_unu FROM user_tables
  WHERE TABLE_NAME = v_tabela;
  RETURN TRUE;
EXCEPTION
WHEN no_data_found THEN
  RETURN FALSE;
END f_exista_tabela;

---------------------------------------------------------------------
FUNCTION f_exista_tabela(tabela_ VARCHAR2) RETURN boolean IS v_tabela VARCHAR2(100);
v_unu NUMBER(1);
BEGIN
  v_tabela := UPPER(tabela_);
  SELECT 1 INTO v_unu FROM user_tables WHERE TABLE_NAME = v_tabela;
  RETURN TRUE;
EXCEPTION
WHEN no_data_found THEN
  RETURN FALSE;
END f_exista_tabela;

---------------------------------------------------------------
PROCEDURE p_sterge_arhive_fact_luna (an_ NUMBER, luna_ NUMBER)
IS
  v_sir VARCHAR2(15);
  v_sir2 VARCHAR2(2000);
  TYPE tRefCursor IS REF CURSOR;
  vRefCursor tRefCursor;
  rec_lf liniifact%ROWTYPE;
BEGIN
      v_sir := '_' || an_ || '_' || luna_;
  -- pentru a nu pierde inregistrarile arhivate le readaugam la tabelele "curente"
  EXECUTE IMMEDIATE 'ALTER TRIGGER trg_facturi_ins DISABLE';
  EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_ins DISABLE';

  IF f_exista_tabela ('FACTURI', an_, luna_) THEN
    EXECUTE IMMEDIATE 'INSERT INTO facturi SELECT * FROM facturi'||v_sir||
      ' WHERE NrFact NOT IN (SELECT NrFact FROM facturi)';
  END IF;
  IF f_exista_tabela ('INCASARI', an_, luna_) THEN
    EXECUTE IMMEDIATE 'INSERT INTO incasari SELECT * FROM incasari'||v_sir||
      ' WHERE CodInc NOT IN (SELECT CodInc FROM incasari)';
  END IF;
  IF f_exista_tabela ('LINIIFACT', an_, luna_) THEN
    v_sir2 := 'SELECT * FROM liniifact' || v_sir || ' WHERE (NrFact, Linie) NOT IN '
      || ' (SELECT NrFact, Linie FROM liniifact ) ORDER BY NrFact, Linie ';
    OPEN vrefcursor FOR v_sir2;
    LOOP
      FETCH vRefCursor INTO rec_lf;
      EXIT WHEN vRefCursor%NOTFOUND;
      INSERT INTO liniifact VALUES rec_lf ;
    END LOOP;
    CLOSE vrefcursor;
   END IF;
  IF f_exista_tabela ('INCASFACT', an_, luna_) THEN
    EXECUTE IMMEDIATE 'INSERT INTO incasfact SELECT * FROM incasfact'||v_sir||
    ' WHERE (CodInc, NrFact) NOT IN (SELECT CodInc, NrFact FROM incasfact)';
  END IF;
```

```
 EXECUTE IMMEDIATE 'ALTER TRIGGER trg_facturi_ins ENABLE';
 EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_ins ENABLE';

    FOR rec_nume IN (SELECT table_name FROM user_tables
         WHERE table_name IN ( 'INCASFACT' || v_sir, 'LINIIFACT' || v_sir,
              'INCASARI' || v_sir, 'FACTURI' || v_sir) ) LOOP
         EXECUTE IMMEDIATE 'DROP TABLE ' || rec_nume.table_name ;
    END LOOP;
END p_sterge_arhive_fact_luna;

--------------------------------------------------------------------------------
PROCEDURE p_creare_arhive_fact_luna (an_ NUMBER, luna_ NUMBER)
IS
    v_sir VARCHAR2(15);
    v_tabela VARCHAR2(100);
BEGIN
    p_sterge_arhive_fact_luna (an_, luna_);

    v_tabela := 'FACTURI' || '_' || LTRIM(TO_CHAR(an_, '9999')) || '_' ||
    LTRIM(TO_CHAR(luna_, '99'));
    IF f_exista_tabela ('FACTURI', an_, luna_) THEN
         raise_application_error(-20120,   'A fost facuta deja arhivarea pe luna '
          || luna_ || ', anul ' || an_);
    END IF;

 v_sir := '_' || an_ || '_' || luna_;
    EXECUTE IMMEDIATE 'CREATE TABLE facturi' || v_sir || ' AS SELECT * FROM facturi
         WHERE  EXTRACT (YEAR FROM DataFact) = '|| an_ ||
              ' AND EXTRACT(MONTH FROM DataFact) = ' || luna_ ;
    EXECUTE IMMEDIATE 'CREATE TABLE liniifact' || v_sir ||
    ' AS SELECT * FROM liniifact
         WHERE NrFact IN (SELECT NrFact FROM facturi' || v_sir || ' ) ';

    EXECUTE IMMEDIATE 'CREATE TABLE incasari' || v_sir ||
    ' AS SELECT * FROM incasari WHERE
         EXTRACT (YEAR FROM DataInc) = ' || an_ ||
              ' AND EXTRACT(MONTH FROM DataInc) = ' || luna_ ||
    '  UNION SELECT * FROM incasari WHERE CodInc IN (
              SELECT CodInc FROM incasfact WHERE NrFact IN (
                        SELECT NrFact FROM facturi WHERE
              EXTRACT (YEAR FROM DataFact) = ' || an_ ||
              ' AND EXTRACT(MONTH FROM DataFact) = ' || luna_ || '  )) ';

    EXECUTE IMMEDIATE 'CREATE TABLE incasfact' || v_sir ||
    ' AS SELECT * FROM incasfact
         WHERE NrFact IN (SELECT NrFact FROM facturi' || v_sir || ') ';

 EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del1 DISABLE';
 EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del2 DISABLE';
 EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del3 DISABLE';

 EXECUTE IMMEDIATE 'DELETE FROM incasfact WHERE (CodInc, NrFact) IN '
   || '(SELECT CodInc, NrFact FROM incasfact' || v_sir || ')';
 EXECUTE IMMEDIATE 'DELETE FROM liniifact WHERE (NrFact, Linie) IN '
   || '(SELECT NrFact, Linie FROM liniifact' || v_sir || ') ';
 EXECUTE IMMEDIATE 'DELETE FROM facturi WHERE NrFact IN '
   || '(SELECT NrFact FROM facturi' || v_sir ||
   ') AND ValTotala = ValIncasata AND NrFact NOT IN (SELECT NrFact FROM incasfact) ';
```

```
  EXECUTE IMMEDIATE 'DELETE FROM incasari WHERE CodInc IN (SELECT CodInc FROM ' ||
    'incasari' || v_sir || ') AND CodInc NOT IN (SELECT CodInc FROM incasfact) ';

  EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del1 ENABLE';
  EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del2 ENABLE';
  EXECUTE IMMEDIATE 'ALTER TRIGGER trg_liniifact_del3 ENABLE';
END p_creare_arhive_fact_luna;

-----------------------------------------------------------------------------------------
FUNCTION f_sir_facturi (data_inceput DATE, data_final DATE) RETURN VARCHAR2
IS
    v_luna_inceput NUMBER(2);
    v_luna_final NUMBER(2);
    v_nume VARCHAR2(30);
    v_clauza_where VARCHAR2(500) := ' ';
    v_sir VARCHAR2(5000) := ' ';
BEGIN
    FOR v_an IN EXTRACT(YEAR FROM data_inceput)..EXTRACT(YEAR FROM data_final)
        LOOP
        IF v_an = EXTRACT(YEAR FROM data_inceput) THEN
            v_luna_inceput := EXTRACT(MONTH FROM data_inceput);
        ELSE
            v_luna_inceput := 1;
        END IF;
        v_luna_final := CASE WHEN v_an = EXTRACT(YEAR FROM data_final)
                    THEN EXTRACT(MONTH FROM data_final) ELSE 12 END;
        FOR v_luna IN v_luna_inceput..v_luna_final LOOP
            v_nume := 'facturi_' || v_an || '_' || v_luna;
            IF f_exista_tabela(v_nume) THEN
                    v_clauza_where := ' WHERE NrFact NOT IN (SELECT NrFact FROM facturi) ';
                    IF v_an = EXTRACT(YEAR FROM data_inceput) AND v_luna =
                                    EXTRACT(MONTH FROM data_inceput) THEN
                            v_clauza_where := v_clauza_where || ' AND DataFact >= DATE'''
                            || data_inceput || '''';
                    END IF;
                    IF v_an = EXTRACT(YEAR FROM data_final) AND v_luna =
                             EXTRACT(MONTH FROM data_final) THEN
                            v_clauza_where := v_clauza_where ||
                            'AND DataFact <= DATE''' || data_final || '''';
                    END IF;
                    v_sir := CASE WHEN v_sir = ' ' THEN v_sir ELSE v_sir || ' UNION ' END
                            || 'SELECT * FROM ' || v_nume || v_clauza_where;
            END IF;
        END LOOP;
    END LOOP;
    v_sir := CASE WHEN v_sir = ' ' THEN v_sir ELSE v_sir || ' UNION ' END
        || ' SELECT * FROM facturi WHERE DataFact BETWEEN DATE''' ||
            data_inceput || '''' || ' AND DATE''' || data_final || '''' ;
    RETURN v_sir;
END f_sir_facturi;
-----------------------------------------------------------------------------------------
PROCEDURE p_fuziune_facturi1(datai DATE, dataf DATE)
IS
  v_sir VARCHAR2(5000) := ' ';
  v_luna_initiala NUMBER(2);
  v_luna_finala NUMBER(2);
  v_an_initial NUMBER(4) := EXTRACT(YEAR FROM datai);
  v_an_final NUMBER(4) := EXTRACT(YEAR FROM dataf);
BEGIN
```

```
     v_sir := f_sir_facturi (datai, dataf);
     --DBMS_OUTPUT.PUT_LINE(v_sir);
     IF pac_arhivare.f_exista_tabela('TEMP_FACTURI') THEN
          EXECUTE IMMEDIATE 'TRUNCATE TABLE TEMP_FACTURI ';
          EXECUTE IMMEDIATE 'DROP TABLE TEMP_FACTURI ';
     END IF;
     EXECUTE IMMEDIATE
          'CREATE GLOBAL TEMPORARY TABLE temp_facturi ON COMMIT '
        || 'PRESERVE ROWS AS SELECT * FROM (' || v_sir || ')  ';

  -- dupa lansarea acestei proceduri, se face un SELECT pe tabela virtuala TEMP_facturi
  -- SELECT * FROM temp_facturi
END p_fuziune_facturi1;

END pac_arhivare;
--=====================================================================
/
```

Lansăm arhivarea pe lunile august, septembrie şi octombrie 2007:

```
BEGIN
  pac_arhivare.p_creare_arhive_fact_luna (2007, 8);
  pac_arhivare.p_creare_arhive_fact_luna (2007, 9);
  pac_arhivare.p_creare_arhive_fact_luna (2007, 10);
END;
```

După cum se poate vedea şi în figura 18.5, chiar şi după arhivare, o parte dintre facturi rămân în continuare în FACTURI (pentru luna septembrie 2007, toate cele şapte facturi). Explicaţia este că aceste facturi comune nu sunt, la momentul arhivării, şi încasate total, deci o să apară în înregistrări ulterioare în ÎNCASFACT.

Enter SQL Statement:

```
SELECT NrFact, DataFact, CodCl, ValTotala, TVA, ValIncasata
FROM facturi_2007_9
  INTERSECT
SELECT NrFact, DataFact, CodCl, ValTotala, TVA, ValIncasata
FROM facturi
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	NRFACT	DATAFACT	CODCL	VALTOTALA	TVA	VALINCASATA
1	3111	01-09-2007	1001	4442959,5	702409,5	0
2	3112	01-09-2007	1005	153442	17142	0
3	3113	02-09-2007	1002	127530	10530	0
4	3115	02-09-2007	1001	110907,5	9157,5	0
5	3116	10-09-2007	1007	136849,5	11299,5	0
6	3117	10-09-2007	1001	287855	33355	0
7	3118	17-09-2007	1001	179565	25065	0

Figura 18.5. Facturi arhivate care se păstrează (încă) şi în tabela principală

Este un motiv cât se poate de serios pentru ca arhivarea pentru o lună să se facă după un timp suficient – 6 luni, 1 an.

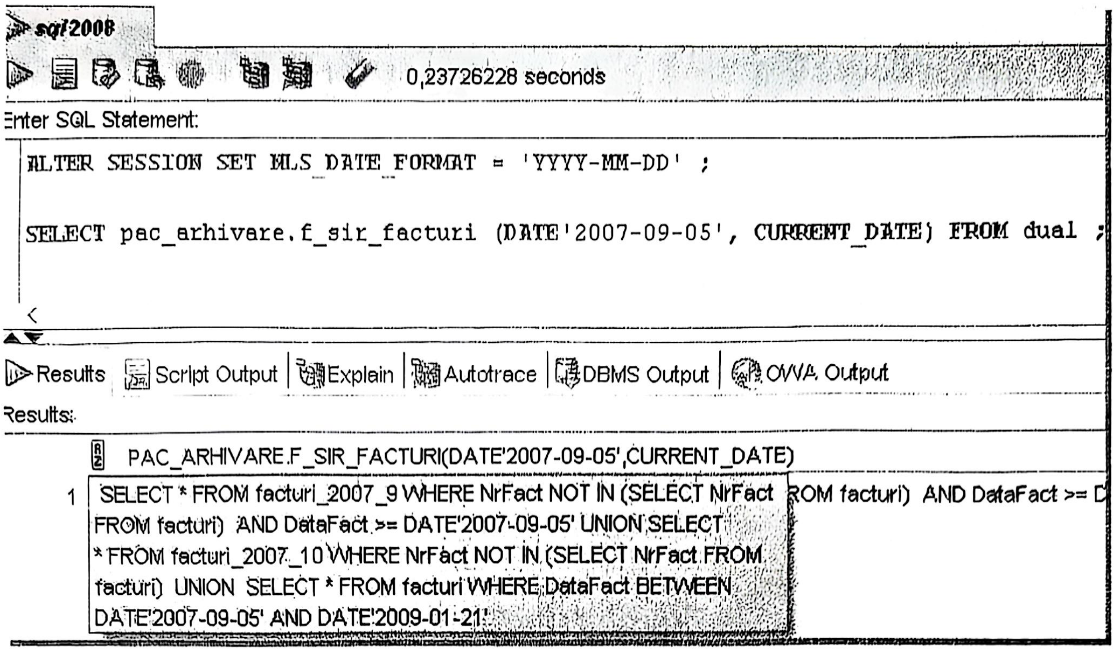


Figura 18.6. Verificarea funcției *pac_arhivare.f_sir_facturi*

În continuare, testăm funcția *f_şir_facturi* pentru a vedea dacă SELECT-ul construit dinamic este corect. Intervalul care interesează este 5 septembrie 2007 - data curentă. Figura 18.6 conține o parte din şir a cărui valoare este, de fapt: *SELECT * FROM facturi_2007_9 WHERE NrFact NOT IN (SELECT NrFact FROM facturi) AND DataFact >= DATE'2007-09-05' UNION SELECT * FROM facturi_2007_10 WHERE NrFact NOT IN (SELECT NrFact FROM facturi) UNION SELECT * FROM facturi WHERE DataFact BETWEEN DATE'2007-09-05' AND DATE'2009-01-21'*. Apelul funcției este precedat de comanda prin care ne asigurăm că formatul datei în sesiunea curentă este YYYY-MM-DD.

În fine, procedura *p_fuziune_facturi1* creează, folosind şirul returnat de funcția *f_şir_facturi* şi SQL dinamic, o tabelă temporară - TEMP_FACTURI - al cărui conținut este privat (accesibil doar în şi din sesiunea curentă). Deranjant este că, de câte ori dorim să afişăm facturile din intervalul specificat, trebuie să lansăm procedura (printr-un bloc anonim) şi apoi să interogăm tabela temporară:

```
begin
  pac_arhivare.p_fuziune_facturi1 (DATE'2007-09-05', CURRENT_DATE);
end ;
```

SELECT * FROM temp_facturi ORDER BY 1;

Din fericire, soluția pare să funcționeze (vezi figura 18.7).

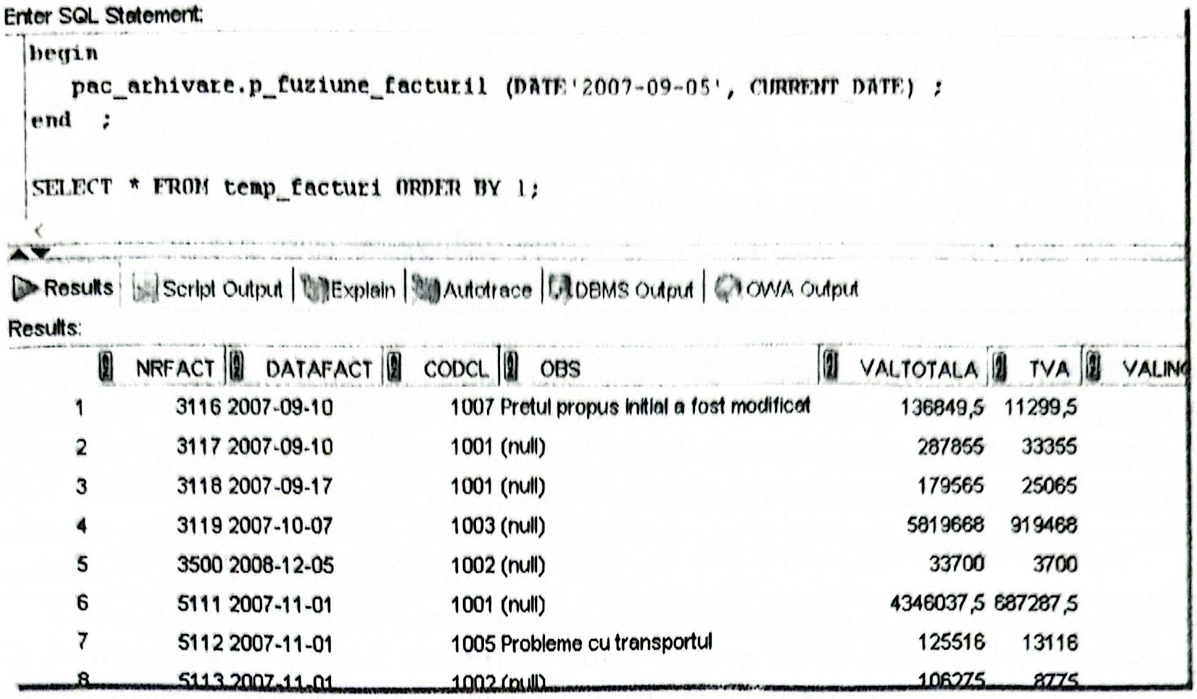


Figura 18.7. Verificarea procedurii *pac_arhivare.p_fuziune_facturi1*

18.3. SQL dinamic în Transact-SQL

Faţă de Oracle PL/SQL, T-SQL este destul de îngăduitor cu multe comenzi DDL care pot fi incluse, fără prea mari probleme, în proceduri şi funcţii. Un argument în acest sens este hipersimpla procedură *p_test* din listingul 18.14, ce creează tabela TEST şi care, la execuţie, nu întâmpină deloc rezistenţă.

Listingul 18.14. Procedură T-SQL prin care se creează o tabelă

```
CREATE PROCEDURE p_test
AS
BEGIN
    CREATE TABLE test (x INTEGER, y VARCHAR(50));
END
```

Înscriindu-ne pe linia comparabilităţii cu cele două paragrafe precedente, redactăm, mai întâi funcţia *f_există_tabela*, funcţie altminteri ajutătoare întrucât nu conţine elemente de SQL dinamic (listingul 18.15). Reamintim că SQL Server este alergic la tipul BOOLEAN, nu numai în tabele, ci şi în proceduri/funcţii, tipul returnat de această funcţie fiind BIT.

Listingul 18.15. Funcţie T-SQL prin care se verifică existenţa unei tabele

```
CREATE FUNCTION f_exista_tabela
    (@tabela_ VARCHAR(40), @an_ SMALLINT, @luna_ TINYINT)
    RETURNS BIT
BEGIN
    DECLARE @v_tabela VARCHAR(40)
    DECLARE @v BIT
    SET @v_tabela = RTRIM(UPPER(@tabela_)) + '_' + CAST (@an_ AS CHAR(4))
        + '_' + RTRIM(CAST (@luna_ AS CHAR(2)))
```

```
    SET @v = (SELECT 1 FROM INFORMATION_SCHEMA.TABLES
            WHERE TABLE_SCHEMA = 'dbo' AND TABLE_TYPE = 'BASE TABLE'
            AND UPPER(TABLE_NAME) = @v_tabela)
    RETURN COALESCE(@v,0)
END
```

Ajungem, astfel, la prima procedură cu urme vizibile de SQL dinamic – *p_şterge_arhive_fact_luna* (listingul 18.16). Ştergerea arhivelor este precedată de re-adăugarea înregistrărilor care, la ştergerea din arhive, s-ar pierde iremediabil, în tabelele „tranzacţionale". Pentru aceasta sunt necesare două operaţiuni. Una dintre ele este comună PostgreSQL şi Oracle, şi anume dezactivarea declanşatoarelor, care se poate face direct în procedură prin comenzile DISABLE/ENABLE TRIGGER. Comanda T-SQL esenţială în materie de SQL dinamic este EXECUTE cu o sintaxă uşor diferită de echivalentele sale din PL/pgSQL şi PL/SQL.

Listingul 18.16. Procedură T-SQL de ştergere a arhivelor lunare legate de facturi

```
CREATE PROCEDURE p_sterge_arhive_fact_luna
    (@an SMALLINT, @luna TINYINT)
AS
BEGIN
    DECLARE @v_sir VARCHAR(15)
    SET @v_sir = '_' + CAST (@an AS CHAR(4)) + '_' + RTRIM(CAST (@luna AS CHAR(2)))

    DECLARE @v_tabela VARCHAR(40)
    DECLARE c_tab CURSOR FOR SELECT table_name
        FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = 'dbo' AND TABLE_TYPE = 'BASE TABLE'
          AND UPPER(TABLE_NAME) IN ( 'INCASFACT' + @v_sir, 'LINIIFACT' + @v_sir,
            'INCASARI' + @v_sir, 'FACTURI' + @v_sir);

    -- pentru a nu pierde inregistrarile arhivate le readaugam la tabelele "curente"
    DISABLE TRIGGER dbo.trg_facturi_ins ON dbo.facturi;
    DISABLE TRIGGER dbo.trg_liniifact_ins ON dbo.liniifact;

    IF dbo.f_exista_tabela ('FACTURI', @an, @luna) = 1
        EXECUTE ('INSERT INTO facturi SELECT * FROM facturi' + @v_sir +
        'WHERE NrFact NOT IN (SELECT NrFact FROM facturi)' );

    IF dbo.f_exista_tabela ('INCASARI', @an, @luna) = 1
        EXECUTE ('SET IDENTITY_INSERT dbo.incasari ON; ' +
        'INSERT INTO incasari (CodInc, DataInc, CodDoc, NrDoc, DataDoc) ' +
        'SELECT *  FROM incasari' + @v_sir + ' WHERE CodInc NOT IN ' +
        '(SELECT CodInc FROM incasari); SET IDENTITY_INSERT dbo.incasari OFF;');

    IF dbo.f_exista_tabela ('LINIIFACT', @an, @luna) = 1
        EXECUTE ('INSERT INTO liniifact SELECT * FROM liniifact' + @v_sir +
            'WHERE CAST (NrFact AS CHAR(8)) + CAST(Linie AS CHAR(2)) NOT IN ' +
        '(SELECT CAST (NrFact AS CHAR(8)) + CAST (Linie AS CHAR(2)) FROM liniifact )
        ORDER BY NrFact, Linie ' )

    IF dbo.f_exista_tabela ('INCASFACT', @an, @luna) = 1
       EXECUTE ('INSERT INTO incasfact SELECT * FROM incasfact' + @v_sir +
        'WHERE CAST(CodInc AS CHAR(10)) + CAST(NrFact AS CHAR(6)) NOT IN '+
        '(SELECT CAST (CodInc AS CHAR(10)) + CAST( NrFact AS CHAR(6)) FROM incasfact)' );

    ENABLE TRIGGER dbo.trg_facturi_ins ON dbo.facturi;
```

```
    ENABLE TRIGGER dbo.trg_liniifact_ins ON dbo.liniifact;
    OPEN c_tab
    FETCH NEXT FROM c_tab INTO @v_tabela
    WHILE @@FETCH_STATUS = 0 BEGIN
        EXECUTE ('DROP TABLE dbo.' + @v_tabela)
        FETCH NEXT FROM c_tab INTO @v_tabela
    END
    CLOSE c_tab
    DEALLOCATE c_tab
END
```

A doua operațiune este nouă, întrucât în tabela ÎNCASĂRI valorile atributului CodInc sunt generate automat, ceea ce are urmări serioase la reinserarea înregistrărilor din tabela-arhivă. Pentru a evita acest gen de probleme, se folosește comanda *SET IDENTITY_INSERT dbo.incasari ON*. După posibila readăugare a înregistrărilor, se reactivează declanșatoarele și se trece la regimul de generare automată a valorilor CodInc din ÎNCASĂRI.

Crearea tabelelor-arhivă pentru o lună dată este subiectul procedurii prezentate în listingul 18.17. Ținând seama că, după crearea arhivelor, o parte importantă dintre înregistrările arhivate trebuie șterse din tabelele tranzacționale, este necesară dezactivarea declanșatoarelor de ștergere ale tabelei LINIIFACT (cele ale tabelei ÎNCASFACT nu fost încă redactate).

Listingul 18.17. Procedură T-SQL de creare a arhivelor lunare legate de facturi

```
CREATE PROCEDURE p_creare_arhive_fact_luna
    (@an SMALLINT, @luna TINYINT)
AS
BEGIN
    DECLARE @v_sir VARCHAR(15)
    SET @v_sir = '_' + CAST (@an AS CHAR(4)) + '_' + RTRIM(CAST (@luna AS CHAR(2)))

    EXECUTE dbo.p_sterge_arhive_fact_luna  @an, @luna

    IF dbo.f_exista_tabela ('FACTURI', @an, @luna) = 1 BEGIN
        RAISERROR(N'Luna arhivata deja !',18, 1)
        ROLLBACK TRANSACTION
    END;

    EXECUTE ('SELECT * INTO facturi' + @v_sir + ' FROM facturi ' +
        ' WHERE YEAR(DataFact) = ' + @an + ' AND MONTH(DataFact) = '+ @luna)
    EXECUTE ('SELECT * INTO liniifact' + @v_sir +
    ' FROM liniifact WHERE NrFact IN (SELECT NrFact FROM facturi' + @v_sir +')' )
    EXECUTE ('SELECT * INTO incasari' + @v_sir + ' FROM incasari WHERE ' +
        ' YEAR(DataInc) = ' + @an +' AND MONTH(DataInc) = ' + @luna +
    '  UNION SELECT * FROM incasari WHERE CodInc IN ( ' +
            'SELECT CodInc FROM incasfact WHERE NrFact IN ( ' +
                    'SELECT NrFact FROM facturi WHERE ' +
            'YEAR(DataFact)=' + @an + ' AND MONTH(DataFact) = ' + @luna+ ' ))' )
    EXECUTE ('SELECT * INTO incasfact'+ @v_sir +' FROM incasfact ' +
        'WHERE NrFact IN (SELECT NrFact FROM facturi' + @v_sir + ') ');

    DISABLE TRIGGER dbo.trg_liniifact_del ON dbo.liniifact;

    EXECUTE ('DELETE FROM incasfact WHERE CAST(CodInc AS CHAR(10)) + '
        + 'CAST (NrFact AS CHAR(6)) IN '
        + '(SELECT CAST(CodInc AS CHAR(10))+CAST(NrFact AS CHAR(6)) FROM incasfact'
```

```
        + @v_sir + ')' )
    EXECUTE ('DELETE FROM liniifact WHERE CAST (NrFact AS CHAR(6)) + '
        + 'CAST (Linie AS CHAR(2)) IN (SELECT CAST(NrFact AS CHAR(6)) + '
        + ' CAST (Linie AS CHAR(2)) FROM liniifact' + @v_sir + ') ' )
    EXECUTE ('DELETE FROM facturi WHERE NrFact IN '
        + '(SELECT NrFact FROM facturi' + @v_sir +
        ') AND ValTotala = ValIncasata AND NrFact NOT IN '
        + ' (SELECT NrFact FROM incasfact) ' )
    EXECUTE ('DELETE FROM incasari WHERE CodInc IN (SELECT CodInc FROM ' +
        'incasari' + @v_sir
        + ') AND CodInc NOT IN (SELECT CodInc FROM incasfact) ' );

    ENABLE TRIGGER dbo.trg_liniifact_del ON dbo.liniifact;
END
```

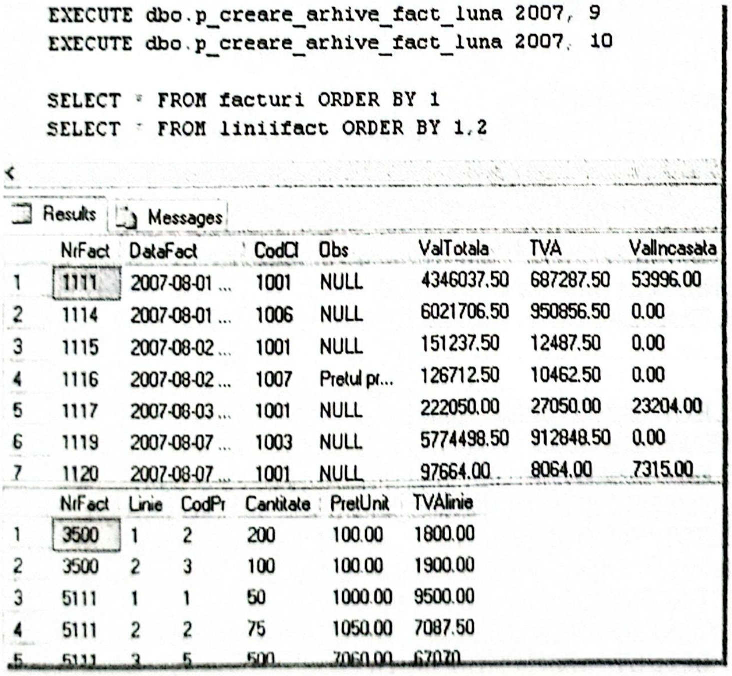
Similar paragrafului anterior, testăm funcţionarea procedurii de arhivare prin execuţia sa pentru lunile august, septembrie şi octombrie 2007:

EXECUTE dbo.p_creare_arhive_fact_luna 2007, 8

EXECUTE dbo.p_creare_arhive_fact_luna 2007, 9

EXECUTE dbo.p_creare_arhive_fact_luna 2007, 10

După arhivarea celor trei luni se observă că în FACTURI au mai rămas facturi (1111, 1114, ...) care nu mai au înregistrări corespondente în LINIIFACT (vezi figura 18.8). Explicaţia este aceeaşi cu cea furnizată pentru Oracle: facturile „rămase" nu sunt încasate în întregime, deci o să apară, în luni următoare celor arhivate, în tabela ÎNCASFACT, deci nu trebuie şterse din tabela-părinte (FACTURI).



```
EXECUTE dbo.p_creare_arhive_fact_luna 2007, 9
EXECUTE dbo.p_creare_arhive_fact_luna 2007, 10

SELECT * FROM facturi ORDER BY 1
SELECT * FROM liniifact ORDER BY 1,2
```

Results | Messages

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	ValIncasata
1	1111	2007-08-01 ...	1001	NULL	4346037.50	687287.50	53996.00
2	1114	2007-08-01 ...	1006	NULL	6021706.50	950856.50	0.00
3	1115	2007-08-02 ...	1001	NULL	151237.50	12487.50	0.00
4	1116	2007-08-02 ...	1007	Pretul pr...	126712.50	10462.50	0.00
5	1117	2007-08-03 ...	1001	NULL	222050.00	27050.00	23204.00
6	1119	2007-08-07 ...	1003	NULL	5774498.50	912848.50	0.00
7	1120	2007-08-07 ...	1001	NULL	97664.00	8064.00	7315.00

	NrFact	Linie	CodPr	Cantitate	PretUnit	TVALinie
1	3500	1	2	200	100.00	1800.00
2	3500	2	3	100	100.00	1900.00
3	5111	1	1	50	1000.00	9500.00
4	5111	2	2	75	1050.00	7087.50
5	5111	3	5	500	7060.00	67070

Figura 18.8. Conţinutul (parţial) al tabelelor FACTURI şi LINIIFACT după arhivare

Întrucât tabelele ÎNCASĂRI şi ÎNCASFACT conţin doar câteva operaţiuni din august 2007, acum conţinutul acestora este gol (vezi figura 18.9), toate înregistrările acestora fiind „mutate" în ÎNCASĂRI_2007_8 şi ÎNCASFACT_2007_8.

```
SELECT * FROM incasari ORDER BY 1
SELECT * FROM incasfact ORDER BY 1,2
SELECT * FROM incasari_2007_8 ORDER BY 1
SELECT * FROM incasfact_2007_8 ORDER BY 1,2
```

Results | Messages

CodInc	DataI...	CodDoc	NrD...	DataDoc

CodInc	NrFact	Transa

	CodInc	DataInc	CodDoc	NrD...	DataDoc
1	1234	2007-08-15 ...	OP	111	2007-08-10 ...
2	1235	2007-08-15 ...	CHIT	222	2007-08-15 ...
3	1236	2007-08-16 ...	OP	333	2007-08-09 ...
4	1237	2007-08-17 ...	CEC	444	2007-08-10 ...
5	1238	2007-08-17 ...	OP	555	2007-08-10 ...
6	1239	2007-08-18 ...	OP	666	2007-08-11 ...

	CodInc	NrFact	Transa
1	1234	1111	53996.00
2	1234	1118	101975.00
3	1235	1112	125516.00
4	1236	1117	9754.00
5	1236	1118	100000.00
6	1236	1120	7315.00
7	1237	1117	9754.00
8	1238	1113	106275.00
9	1239	1117	3696.00

Figura 18.9. Conținutul tabelelor ÎNCASĂRI și ÎNCASFACT, precum și ale arhivelor lor pe luna august 2007

Funcția ce construiește, prin eventuale UNION-uri, întreaga interogare prin care, din arhive și tabela FACTURI, sunt extrase toate facturile dintr-un interval calendaristic specificat este afișată în listingul 18.18. Cum deja ne-am obișnuit, în T-SQL nu există structuri repetitive de tipul *FOR i = 1 TO n...*, așa că se folosește WHILE. O altă problemă a fost conversia constantelor de tip dată calendaristică în șiruri de caractere, operațiune pentru care am folosit funcția CONVERT.

Listingul 18.18. Funcție ce returnează interogarea SQL pentru reconstituirea facturilor

```
CREATE FUNCTION f_sir_facturi
    (@data_inceput SMALLDATETIME, @data_final SMALLDATETIME)
RETURNS VARCHAR(5000)
AS
BEGIN
    DECLARE @v_luna TINYINT
    DECLARE @v_luna_inceput TINYINT
    DECLARE @v_luna_final TINYINT
    DECLARE @v_clauza_where VARCHAR(500)
    DECLARE @v_sir VARCHAR(5000)
    DECLARE @v_nume VARCHAR(50)
    SET @v_sir = ''
    DECLARE @v_an SMALLINT
    SET @v_an = YEAR(@data_inceput)
    SET @v_clauza_where = ''

    WHILE @v_an <= YEAR(@data_final) BEGIN

        IF @v_an = YEAR(@data_inceput)
```

```
            SET @v_luna_inceput = MONTH(@data_inceput)
        ELSE
            SET @v_luna_inceput = 1

        IF @v_an = YEAR(@data_final)
            SET @v_luna_final = MONTH(@data_final)
        ELSE
            SET @v_luna_final = 12

        SET @v_luna = @v_luna_inceput

        WHILE @v_luna <= @v_luna_final BEGIN
            SET @v_nume = 'facturi_' + CAST( @v_an AS CHAR(4)) + '_' +
                    RTRIM(CAST (@v_luna AS CHAR(2)))

            IF dbo.f_exista_tabela ('facturi', @v_an, @v_luna) = 1 BEGIN
                    SET @v_clauza_where =
                            'WHERE NrFact NOT IN (SELECT NrFact FROM facturi ) '

                    IF @v_an = YEAR(@data_inceput) AND @v_luna = MONTH(@data_inceput)
                            SET @v_clauza_where = @v_clauza_where + ' AND DataFact >= ''' +
                                    CONVERT (CHAR(10), @data_inceput, 102) + ''''

                    IF @v_an = YEAR(@data_final) AND @v_luna = MONTH(@data_final)
                            SET @v_clauza_where = @v_clauza_where + ' AND DataFact <= ''' +
                                    CONVERT (CHAR(10), @data_final, 102) + ''''

                    SET @v_sir = CASE WHEN @v_sir = ' ' THEN @v_sir ELSE @v_sir +
                            ' UNION ' END + 'SELECT * FROM ' + @v_nume + @v_clauza_where

            END
            SET @v_luna = @v_luna + 1
        END

        SET @v_an = @v_an + 1
        END

    SET @v_sir = CASE WHEN @v_sir = ' ' THEN @v_sir ELSE @v_sir + ' UNION ' END
        + ' SELECT * FROM facturi WHERE DataFact BETWEEN ''' +
            CONVERT (CHAR(10), @data_inceput, 102) + '''' + ' AND ''' +
            CONVERT (CHAR(10), @data_final, 102)+ ''''

    RETURN @v_sir
END
```

Testarea funcţiei poate fi făcută în SQL Server şi cu PRINT (am avut, de fapt, nevoie de o opţiune pentru afişarea unui şir lung de caractere pe mai multe linii):

PRINT dbo.f_sir_facturi ('2007-08-05', GETDATE())

şirul fiind conform cu aşteptările (vezi figura 18.10).

```
PRINT dbo.f_sir_facturi ('2007-08-05' GETDATE()
```

Messages

```
SELECT * FROM facturi_2007_8 WHERE NrFact NOT IN (SELECT NrFact FROM facturi )
  AND DataFact >= '2007.08.05'
UNION SELECT * FROM facturi_2007_9 WHERE NrFact NOT IN (SELECT NrFact FROM facturi )
UNION SELECT * FROM facturi_2007_10 WHERE NrFact NOT IN (SELECT NrFact FROM facturi )
UNION  SELECT * FROM facturi WHERE DataFact BETWEEN '2007.08.05' AND '2009.01.21'
```

Figura 18.10. Conținutul tabelelor ÎNCASĂRI și ÎNCASFACT, precum și ale arhivelor lor pe luna august 2007

Procedura care creează tabela temporară globală este una destul de simplă (vezi listingul 18.19). Numele tabelei temporare este specificat la invocare de către utilizator (vezi figura 18.11):

EXECUTE dbo.p_fuziune_facturi '2007-09-05', '2007-11-10', 'fact_temp2'

SELECT * FROM dbo.##fact_temp2 ORDER BY 1

Listingul 18.19. Procedura T-SQL care creează o tabelă temporară ce reconstituie facturile

```
CREATE PROCEDURE p_fuziune_facturi
    (@datai SMALLDATETIME, @dataf SMALLDATETIME, @nume_tabela VARCHAR(30))
AS
BEGIN
    DECLARE @v_sir VARCHAR(5000)
    SET @v_sir = dbo.f_sir_facturi (@datai, @dataf)
    EXECUTE ('SELECT * INTO ##' + @nume_tabela + ' FROM (' +@v_sir +') t ')
END
```

```
EXECUTE dbo.p_fuziune_facturi '2007-09-05', '2007-11-10', 'fact_temp2'
SELECT * FROM dbo.##fact_temp2 ORDER BY 1
```

Results | Messages

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA
1	3116	2007-09-10 00:00:00	1007	Pretul propus initial a fost modificat	136849.50	11299.50
2	3117	2007-09-10 00:00:00	1001	NULL	287855.00	33355.00
3	3118	2007-09-17 00:00:00	1001	NULL	179565.00	25065.00
4	3119	2007-10-07 00:00:00	1003	NULL	5819668.00	919468.00
5	3120	2007-10-01 00:00:00	1001	NULL	571200.00	91200.00
6	5111	2007-11-01 00:00:00	1001	NULL	4346037.50	687287.50
7	5112	2007-11-01 00:00:00	1005	Probleme cu transportul	125516.00	13116.00
8	5113	2007-11-01 00:00:00	1002	NULL	106275.00	8775.00
9	5114	2007-11-01 00:00:00	1006	NULL	6021706.50	950856.50
10	5115	2007-11-02 00:00:00	1001	NULL	151237.50	12487.50
11	5116	2007-11-02 00:00:00	1007	Pretul propus initial a fost modificat	126712.50	10462.50
12	5117	2007-11-03 00:00:00	1001	NULL	222050.00	27050.00
13	5118	2007-11-04 00:00:00	1001	NULL	201975.00	29475.00
14	5119	2007-11-07 00:00:00	1003	NULL	5774498.50	912848.50
15	5120	2007-11-07 00:00:00	1001	NULL	97664.00	8064.00
16	5121	2007-11-07 00:00:00	1004	NULL	4737838.00	747638.00

Figura 18.11. Crearea și apelul tabelei temporare ##TEMP_FACT

18.4. SQL dinamic în IBM DB2 SQL PL

Oracle PL/SQL rămâne singurul limbaj dintre cele patru discutate în această lucrare care nu permite crearea unei tabele prin program, întrucât procedura simplă redactată în IBM DB2 SQL PL din listingul 18.20 nu întâmpină rezistenţă la prima sa lansare în execuţie.

Listingul 18.20. O (banală) procedură SQL PL prin care se creează o tabelă

```
CREATE PROCEDURE p_test ( )
P1: BEGIN
    CREATE TABLE test (x INTEGER, y VARCHAR(50));
END P1
```

Nici în SQL PL, funcţia *f_există_tabela* nu are legătură cu SQL dinamic (listingul 18.21). Similar SQL Server, SQL PL este destul de irascibil la tipul BOOLEAN, chiar şi în proceduri şi funcţii (în acest caz, DB2 nu acceptă clauza *RETURNS BOOLEAN*). Aşa că tipul returnat de funcţie este SMALLINT.

Listingul 18.21. Funcţie SQL PL prin care se verifică existenţa unei tabele

```
CREATE FUNCTION f_exista_tabela(tabela_ VARCHAR(40), an_ SMALLINT, luna_ SMALLINT)
    RETURNS SMALLINT
F1: BEGIN ATOMIC
    DECLARE v_tabela VARCHAR(40);
    DECLARE v SMALLINT;
    SET v_tabela = RTRIM(UPPER(tabela_)) || '_' || CAST (an_ AS CHAR(4))
        || '_' || RTRIM(CAST (luna_ AS CHAR(2)));

    SET v = (SELECT 1 FROM SYSCAT.tables
        WHERE tabschema = CURRENT_USER AND type = 'T' AND UPPER(tabname) = v_tabela );
    RETURN COALESCE(v,0);
END
```

Noutatea majoră căreia trebuie să-i găsim o soluţie în DB2 este faptul că declanşatoarele nu pot fi dezactivate şi reactivate. În aceste condiţii, orice inserare în LINIIFACT incrementează ValTotală şi TVA în FACTURI, iar orice inserare în ÎNCASFACT incrementează ValÎncasată din FACTURI. La fel (dar cu semn schimbat) stau lucrurile şi la ştergeri de înregistrări din LINIIFACT şi ÎNCASFACT. Renunţăm la procedura de ştergere a arhivelor şi mutăm o parte din operaţiunile sale în procedura *p_creare_arhive_fact_luna* (vezi listingul 18.22). Prima porţiune a procedurii verifică dacă există arhivă pe anul şi luna date. Dacă da, atunci se şterg din arhivă numai înregistrările care sunt în tabela principală corespondentă. Dacă nu există arhiva, se creează folosind sintaxa *CREATE TABLE tabela_an_luna LIKE tabela*. După verificarea existenţei arhivei şi crearea sa sau ştergerea unora dintre înregistrările sale, în arhivă se inserează înregistrările din tabela principală ce corespund anului şi lunii curente.

Listingul 18.22. Varianta SQL PL a procedurii de arhivare lunară a facturilor/incasărilor

```
CREATE PROCEDURE p_creare_arhive_fact_luna
    (an_ SMALLINT, luna_ SMALLINT)
P1: BEGIN
    DECLARE v_exec VARCHAR(2000);
    DECLARE v_sir VARCHAR(15);
    SET v_sir = '_' || CAST (an_ AS CHAR(4)) || '_' || RTRIM(CAST (luna_ AS CHAR(2)));

    --- creare arhive sau stergerea unora dintre inregistrarile acestora (ce exista in tabelele principale)
    IF f_exista_tabela ('FACTURI', an_, luna_) = 1 THEN
        SET v_exec = 'DELETE FROM facturi' || v_sir ||
            'WHERE NrFact IN (SELECT NrFact FROM facturi)';
        EXECUTE IMMEDIATE v_exec;
    ELSE
        SET v_exec = 'CREATE TABLE facturi' || v_sir || ' LIKE facturi ';
        EXECUTE IMMEDIATE v_exec;
    END IF;
    SET v_exec = 'INSERT INTO facturi' || v_sir || ' SELECT * FROM facturi ' ||
        'WHERE YEAR(DataFact)= ' || CAST (an_ AS CHAR(4)) ||
        'AND MONTH(DataFact)=' || CAST (luna_ AS CHAR(2));
    EXECUTE IMMEDIATE v_exec;

    IF f_exista_tabela ('LINIIFACT', an_, luna_) = 1 THEN
        SET v_exec = 'DELETE FROM liniifact' || v_sir ||
            ' WHERE (NrFact, Linie) IN (SELECT NrFact, Linie FROM liniifact)';
        EXECUTE IMMEDIATE v_exec;
    ELSE
        SET v_exec = 'CREATE TABLE liniifact' || v_sir || ' LIKE liniifact ';
        EXECUTE IMMEDIATE v_exec;
    END IF;
    SET v_exec = 'INSERT INTO liniifact' || v_sir ||
    ' SELECT * FROM liniifact WHERE NrFact IN (SELECT NrFact FROM facturi'|| v_sir || ')';
    EXECUTE IMMEDIATE v_exec;

    IF f_exista_tabela ('INCASARI', an_, luna_) = 1 THEN
        SET v_exec = 'DELETE FROM incasari' || v_sir ||
            ' WHERE CodInc IN (SELECT CodInc FROM incasari)';
        EXECUTE IMMEDIATE v_exec;
    ELSE
        SET v_exec = 'CREATE TABLE incasari' || v_sir || ' LIKE incasari ';
        EXECUTE IMMEDIATE v_exec;
    END IF;
    SET v_exec = 'INSERT INTO incasari' || v_sir || ' SELECT * FROM incasari WHERE ' ||
        ' YEAR(DataInc) = ' || CAST (an_ AS CHAR(4)) ||
        ' AND MONTH(DataInc) = ' || CAST (luna_ AS CHAR(2)) ||
    '  UNION SELECT * FROM incasari WHERE CodInc IN ( ' ||
            ' SELECT CodInc FROM incasfact WHERE NrFact IN ( ' ||
                    'SELECT NrFact FROM facturi WHERE  ' ||
            ' YEAR(DataFact) = ' || CAST (an_ AS CHAR(4)) || ' AND MONTH(DataFact) = ' ||
            CAST (luna_ AS CHAR(2)) || ' )) ';
    EXECUTE IMMEDIATE v_exec ;

    IF f_exista_tabela ('INCASFACT', an_, luna_) = 1 THEN
        SET v_exec = 'DELETE FROM incasfact' || v_sir ||
            ' WHERE (CodInc, NrFact) IN (SELECT CodInc, NrFact FROM incasfact)' ;
        EXECUTE IMMEDIATE v_exec ;
    ELSE
        SET v_exec = 'CREATE TABLE incasfact' || v_sir || ' LIKE incasfact ' ;
```

```
        EXECUTE IMMEDIATE v_exec ;
    END IF;
    SET v_exec = 'INSERT INTO incasfact' ||  v_sir || ' SELECT * FROM incasfact ' ||
        ' WHERE NrFact IN (SELECT NrFact FROM facturi' || v_sir ||
            ') AND CodInc IN (SELECT CodInc FROM incasari' || v_sir || ') ';
    EXECUTE IMMEDIATE v_exec ;

    --- urmeaza arhivarea propriu-zisa; din tabelele principale se sterg numai datele
    --- facturilor incasate complet

    SET v_exec = 'DELETE FROM incasfact WHERE (CodInc, NrFact) IN '
        || '(SELECT CodInc, NrFact FROM incasfact' || v_sir ||
        ') AND NrFact IN (SELECT NrFact FROM facturi WHERE ValTotala = ValIncasata )';
    EXECUTE IMMEDIATE v_exec;

    SET v_exec = 'DELETE FROM liniifact WHERE (NrFact,Linie) IN (' ||
        ' SELECT NrFact,Linie FROM liniifact' || v_sir || ') AND NrFact IN (' ||
        ' SELECT NrFact FROM facturi WHERE ValTotala = ValIncasata )';
    EXECUTE IMMEDIATE v_exec;

    SET v_exec = 'DELETE FROM facturi WHERE NrFact IN '
        || '(SELECT NrFact FROM facturi' || v_sir ||
        ') AND ValTotala = ValIncasata ';
    EXECUTE IMMEDIATE v_exec;

    SET v_exec = 'DELETE FROM incasari WHERE CodInc IN (SELECT CodInc FROM ' ||
        'incasari' || v_sir
        || ') AND CodInc NOT IN (SELECT CodInc FROM incasfact) ';
    EXECUTE IMMEDIATE v_exec;

END P1
```

După arhivare se şterg numai înregistrări care se referă la facturi încasate în totalitate. Se creează arhivele pe lunile august, septembrie şi octombrie 2007:

CALL p_creare_arhive_fact_luna (2007,8)

CALL p_creare_arhive_fact_luna (2007,9)

CALL p_creare_arhive_fact_luna (2007,10)

Listingul 18.22 conţine funcţia SQL PL pentru recompunerea facturilor din tabelele curente şi din arhive pentru un interval dat.

Listingul 18.23. Variantă SQL PL a funcţiei de reconstituire a facturilor arhivate

```
CREATE FUNCTION f_sir_facturi (data_inceput DATE, data_final DATE)
    RETURNS VARCHAR(5000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_luna SMALLINT;
    DECLARE v_luna_inceput SMALLINT;
    DECLARE v_luna_final SMALLINT;
    DECLARE v_clauza_where VARCHAR(500);
    DECLARE v_sir VARCHAR(5000);
    DECLARE v_nume VARCHAR(50);
    DECLARE v_an SMALLINT;
    SET v_sir = ' ';
    SET v_an = YEAR(data_inceput);
    SET v_clauza_where = ' ';
```

```
    WHILE (v_an <= YEAR(data_final)) DO

        IF v_an = YEAR(data_inceput) THEN
            SET v_luna_inceput = MONTH(data_inceput);
        ELSE
            SET v_luna_inceput = 1;
        END IF;

        IF v_an = YEAR(data_final)THEN
            SET v_luna_final = MONTH(data_final);
        ELSE
            SET v_luna_final = 12;
        END IF;

        SET v_luna = v_luna_inceput;
        WHILE (v_luna <= v_luna_final) DO

            SET v_nume = 'facturi_' || CAST( v_an AS CHAR(4)) || '_' ||
                    RTRIM(CAST (v_luna AS CHAR(2)));
            IF f_exista_tabela ('facturi', v_an, v_luna) = 1 THEN
                    SET v_clauza_where = ' WHERE NrFact NOT IN (SELECT NrFact FROM facturi) ';
                    IF v_an = YEAR(data_inceput) AND v_luna = MONTH(data_inceput) THEN
                            SET v_clauza_where = v_clauza_where || ' AND DataFact >= ''' ||
                                        CAST (data_inceput AS CHAR(10)) || '''';
                    END IF;
                    IF v_an = YEAR(data_final) AND v_luna = MONTH(data_final) THEN
                            SET v_clauza_where = v_clauza_where || ' AND DataFact <= ''' ||
                                    CAST (data_final AS CHAR(10)) || '''';
                    END IF;
                    SET v_sir = CASE WHEN v_sir = ' ' THEN v_sir ELSE v_sir || ' UNION ' END
                            || 'SELECT * FROM ' || v_nume || v_clauza_where;
            END IF;
            SET v_luna = v_luna + 1;
        END WHILE;
        SET v_an = v_an + 1;
    END WHILE;

    SET v_sir = CASE WHEN v_sir = ' ' THEN v_sir ELSE v_sir || ' UNION ' END
        || ' SELECT * FROM facturi WHERE DataFact BETWEEN ''' ||
            CAST (data_inceput AS CHAR(10)) || '''' || ' AND ''' ||
            CAST (data_final AS CHAR(10)) || '''';

    RETURN v_sir;

END
```

În fine, ca şi în T-SQL, procedura din listingul 18.24 creează o tabelă temporară globală cu un nume indicat la apelare, tabelă temporară ce conţine facturile din intervalul calendaristic specificat prin *data_iniţială* şi *data_finală*.

Listingul 18.24. Procedură SQL PL ce creează o tabelă temporară globală

```
CREATE PROCEDURE p_fuziune_facturi1 (data_initiala DATE,
            data_finala DATE, tab_temporara VARCHAR(30))
P1: BEGIN
    DECLARE v_sir VARCHAR(5000);

    SET v_sir = 'DECLARE GLOBAL TEMPORARY TABLE session.' || tab_temporara ||
        'LIKE facturi ON COMMIT PRESERVE ROWS ';
```

```
    EXECUTE IMMEDIATE v_sir;

    SET v_sir = ' INSERT INTO session.' || tab_temporara || ' '
        || f_sir_facturi (data_initiala, data_finala);
    EXECUTE IMMEDIATE v_sir;

END P1
```

Tot similar T-SQL, apelăm procedura, numele indicat al tabelei temporare fiind TAB_TEMP3 şi se obţin înregistrările din figura 18. 12.

CALL p_fuziune_facturi1 (CAST ('2007-09-05' AS DATE),

CAST ('2007-11-10' AS DATE), 'tab_temp3')

SELECT * FROM session.tab_temp3 ORDER BY 1

```
CALL p_fuziune_facturi1 (CAST ('2007-09-05' AS DATE), CAST ('2007-11-10' AS DATE), 'tab_temp3')

SELECT * FROM session.tab_temp3 ORDER BY 1

------------------------------ Commands Entered ------------------------------
SELECT * FROM session.tab_temp3 ORDER BY 1;
-----------------------------------------------------------------------------

SELECT * FROM session.tab_temp3 ORDER BY 1

NRFACT      DATAFACT   CODCL OBS                                               VALTOTALA    TVA          VAL
----------- ---------- ----- -------------------------------------------------- ------------ ------------ ---
       3116 09/10/2007  1007 Pretul propus initial a fost modificat              136849.50     11299.50
       3117 09/10/2007  1001 -                                                   287855.00     33355.00
       3118 09/17/2007  1001 -                                                   179565.00     25065.00
       3119 10/07/2007  1003 -                                                  5819668.00    919468.00
       5111 11/01/2007  1001 -                                                  4346037.50    687287.50
       5112 11/01/2007  1005 Probleme cu transportul                             125516.00     13116.00
       5113 11/01/2007  1002 -                                                   106275.00      8775:00
       5114 11/01/2007  1006 -                                                  6021706.50    950856.50
       5115 11/02/2007  1001 -                                                   151237.50     12487.50
       5116 11/02/2007  1007 Pretul propus initial a fost modificat              126712.50     10462.50
       5117 11/03/2007  1001 -                                                   222050.00     27050.00
       5118 11/04/2007  1001 -                                                   201975.00     29475.00
       5119 11/07/2007  1003 -                                                  5774498.50    912848.50
       5120 11/07/2007  1001 -                                                    97664.00      8064.00
       5121 11/07/2007  1004 -                                                  4737838.00    747638.00
       5122 11/07/2007  1005 -                                                           -            -

  16 record(s) selected.
```

Figura 18.12. Crearea şi apelul tabelei temporare TAB_TEMP3

Încheiem paragraful cu datoria din capitolul anterior legată de furnizarea unui mecanism minimal de protejare a atributelor calculate de modificări interactive. Spuneam atunci că există în DB2 variabile globale, însă acestea nu pot fi modificate din declanşatoare, aşa că suntem nevoiţi să găsim o altă soluţie. Soluţia pe care o discutăm acum se bazează pe o tabelă temporară (care se creează automat în schema SESSION) prin comanda DECLARE GLOBAL TEMPORARY TABLE. Tabela temporară îşi poate păstra schema şi conţinutul doar pe perioada unei sesiuni de lucru cu baza de date, conţinutul său fiind privat. Necazul este că orice procedură, funcţie şi declanşator ce face referire la tabela temporară o să aibă necazuri mari la compilare, întrucât este vorba despre un obiect nepersistent în schema bazei.

Tocmai aici îşi arată o parte dintre virtuţi SQL-ul dinamic. Redactăm, pentru început, o procedură (vezi listingul 18.25), care verifică existenţa tabelei temporare

în schema SESSION. Dacă nu există, o creează şi îi inserează o linie. Tabela se numeşte V_TRG_LINIIFACT şi conţine un atribut X care va avea două valori, 0 şi 1.

Listingul 18.25. Procedură SQL PL ce creează o tabelă temporară globală

```
CREATE PROCEDURE p_creare_v_trg_liniifact ()
P1: BEGIN
    DECLARE v_sir VARCHAR(500);
    DECLARE v SMALLINT;
    DECLARE v2 SMALLINT;
    DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
    DECLARE SQLCODE INT DEFAULT 0;

  DECLARE c_notfound CONDITION FOR SQLSTATE '42704';
    DECLARE CONTINUE HANDLER FOR c_notfound SET v2 = 1;

    SET v_sir = 'UPDATE SESSION.V_TRG_LINIIFACT SET x = x ';
    EXECUTE IMMEDIATE v_sir;
  IF v2 = 1 THEN
    SET v_sir = 'DECLARE GLOBAL TEMPORARY TABLE v_trg_liniifact (x SMALLINT ) ON COMMIT
PRESERVE ROWS ';
    EXECUTE IMMEDIATE v_sir;
    SET v_sir = 'INSERT INTO session.v_trg_liniifact VALUES (0)';
    EXECUTE IMMEDIATE v_sir;
  END IF;
 END P1
```

O a doua procedură, cea din listingul 18.26, modifică valoarea atributului X din tabela temporară. Pentru a evita eroarea ce s-ar declanşa la apelul acestei proceduri fără crearea prealabilă a tabelei, comanda UPDATE este precedată de un apel la procedura anterioară.

Listingul 18.26. Procedură SQL PL ce setează valoarea atributului X în tabela temporară

```
CREATE PROCEDURE p_setare_v_trg_liniifact ( valoare_ SMALLINT)
P1: BEGIN
    DECLARE v_sir VARCHAR(500);
    CALL p_creare_v_trg_liniifact;
    SET v_sir = 'UPDATE session.v_trg_liniifact SET x = ' || CAST (valoare_ AS CHAR(1));
    EXECUTE IMMEDIATE v_sir;
END P1
```

Ceva mai mult de lucru am avut cu o funcţie care trebuie să returneze valoarea curentă a atributului X din singura linie a tabelei temporare. Cum n-am fost în stare să o redactez, am apelat la o procedură altminteri simplă (vezi listingul 18.27), în care utilizăm un parametru de tip OUT (*v_*). Procedura foloseşte un cursor dinamic care încarcă singura înregistrare din tabela temporară în parametrul de ieşire.

Listingul 18.27. Procedură SQL PL ce preia valoarea atributului X din tabela temporară

```
CREATE PROCEDURE p_get_v_trg_liniifact (OUT v_ SMALLINT)
P1: BEGIN
   DECLARE v_sir      VARCHAR(200);
   DECLARE c_x CURSOR FOR v_cur_stmt;
   SET v_sir = ' SELECT x FROM session.v_trg_liniifact ';
   PREPARE v_cur_stmt FROM v_sir;
```

```
   OPEN c_x;
   FETCH c_x INTO v_;
  CLOSE c_x;
END P1
```

Nu ne mai rămâne decât să modificăm declanşatoarele tabelelor LINIIFACT şi FACTURI. În DB2, modificarea declanşatoarelor nu este posibilă, aşa că mai întâi le ştergem şi apoi compilăm noile versiuni.

DROP TRIGGER trg_liniifacturi_ins2

Declanşatorul de tip BEFORE INSERT (trg_linii_fact_ins1) rămâne neschimbat. Cel de-al doilea declanşator (vezi listingul 18.28) apelează înainte şi după actualizarea atributelor calculate din tabela FACTURI, procedura prin care se setează valoarea atributului X în tabela temporară. Setarea se face pe 1 înainte de UPDATE şi pe 0 după UPDATE. Scopul este clar: valoarea atributului X din V_TRG_LINIIFACT va fi 1 doar pe perioada actualizării celor două atribute ale tabelei FACTURI.

Listingul 18.28. Noua variantă a celui de-al doilea declanşator de inserare în LINIIFACT

```
-- NOUL declansatorul la nivel de linie de tipul AFTER
CREATE TRIGGER trg_liniifacturi_ins2
AFTER INSERT ON liniifact
REFERENCING NEW AS NEW FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    -- declansatorul TRG_LINIIFACT_INS1 ramane neschimbat

    CALL p_setare_v_trg_liniifact (1);

    UPDATE facturi
    SET tva = COALESCE(tva,0) + NEW.tvalinie,
        valtotala = COALESCE(valtotala,0) + NEW.cantitate * NEW.pretunit + NEW.tvalinie
    WHERE NrFact = NEW.NrFact;

    CALL p_setare_v_trg_liniifact (0);
END
```

Şi declanşatorul de modificare a unei înregistrări din FACTURI trebuie actualizat. Mai întâi scăpăm de versiunea curentă:

DROP TRIGGER trg_facturi_upd_a_r_nrfact

şi apoi îl compilăm pe cel din listingul 18.29, care are un nume uşor schimbat.

Listingul 18.29. Noul declanşator de modificare pentru FACTURI

```
CREATE TRIGGER trg_facturi_upd_a_r
AFTER UPDATE OF NrFact, ValTotala, TVA, ValIncasata ON facturi
REFERENCING OLD AS OLD  NEW AS NEW FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    DECLARE v_vp SMALLINT DEFAULT 0;
    IF OLD.NrFact <> NEW.NrFact THEN
        UPDATE liniifact SET NrFact = NEW.NrFact WHERE NrFact = OLD.NrFact;
        UPDATE incasfact SET NrFact = NEW.NrFact WHERE NrFact = OLD.NrFact;
    END IF;

    IF OLD.tva <> NEW.tva OR OLD.ValTotala <> NEW.ValTotala THEN
        CALL p_get_v_trg_liniifact (v_vp);
        IF COALESCE(v_vp,0) <> 1 THEN
            SIGNAL SQLSTATE '80000'
```

```
        SET MESSAGE_TEXT = 'Nu puteti modifica interactiv valorilor TVA si ValTotala!';
        END IF;
    END IF;
END
```

Testarea o facem inserând două linii într-o factură nouă, cu numărul 4002:

INSERT INTO facturi VALUES (4002, CURRENT_DATE, 1002, NULL, 123456, 234567, 345678, NULL, NULL)

INSERT INTO liniifact VALUES (4002, 1, 2, 200, 100, 8989989);

INSERT INTO liniifact VALUES (4002, 2, 3, 100, 100, 0);

după care încercăm să modificăm direct valoarea atributului ValTotală din FACTURI în linia corespunzătoare facturii 4002:

UPDATE facturi SET ValTotala = 1234567 WHERE NrFact = 4002

Declanşatoarele funcţionează corect, după cum ne confirmă figura 18.13.

```
INSERT INTO facturi VALUES (4002, CURRENT_DATE, 1002, NULL, 123456, 234567, 345678, NULL, NULL)
INSERT INTO liniifact VALUES (4002, 1, 2, 200, 100, 8989989) ;
INSERT INTO liniifact VALUES (4002, 2, 3, 100, 100, 0) ;

UPDATE facturi SET ValTotala = 1234567 WHERE NrFact = 4002

INSERT INTO facturi VALUES (4002, CURRENT_DATE, 1002, NULL, 123456, 234567, 345678, NULL, NULL)
DB20000I  The SQL command completed successfully.

------------------------------ Commands Entered ------------------------------
INSERT INTO liniifact VALUES (4002, 1, 2, 200, 100, 8989989) ;
------------------------------------------------------------------------------
INSERT INTO liniifact VALUES (4002, 1, 2, 200, 100, 8989989)
DB20000I  The SQL command completed successfully.

------------------------------ Commands Entered ------------------------------
INSERT INTO liniifact VALUES (4002, 2, 3, 100, 100, 0) ;
------------------------------------------------------------------------------
INSERT INTO liniifact VALUES (4002, 2, 3, 100, 100, 0)
DB20000I  The SQL command completed successfully.

------------------------------ Commands Entered ------------------------------
UPDATE facturi SET ValTotala = 1234567 WHERE NrFact = 4002;
------------------------------------------------------------------------------
UPDATE facturi SET ValTotala = 1234567 WHERE NrFact = 4002
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0438N  Application raised error with diagnostic text: "Nu puteti modifica
interactiv valorilor TVA si ValTotala !".  SQLSTATE=80000

SQL0438N  Application raised error with diagnostic text: "Nu puteti modifica interactiv valorilor TVA si ValTotala !
```

Figura 18.13. Verificarea procedurilor şi declanşatoarelor ce protejează atributele calculate

Capitolul 19
Obiectual-relațional în SQL

Fuziunea modelelor relațional şi obiectual de organizare a datelor este o temă ce preocupă teoreticienii şi practicienii bazelor de date de la finalul anilor '80. Ca răspuns la cerințele dezvoltatorilor de aplicații şi concurența unor limbaje precum OQL (Object Query Language), versiunea SQL:1999 a standardului a adus noutăți radicale în materie de gestiune a obiectelor în bazele de date relaționale. De la această versiune, putem spune că SQL înglobează elementele esențiale are modelului obiectual-relațional (O-R). Perioada de redactare a primei versiuni obiectual-relaționale a standardului coincide cu cea de eferverscență O-R în rândul producătorilor de SGBD-uri. A doua parte a anilor '90 şi imediat după 2000 reprezintă anii în care producătorii au inclus în serverele BD *lame* şi *cartuşe de date* (*data blades, data cartridges*) – extensii ale tipurilor primitive dedicate stocării şi manipulării de texte, imagini, sunete, hărți de mari dimensiuni. Apoi... linişte. După ce au făcut atâta tam-tam pe seama lipsei opțiunilor orientate pe obiecte (OO) în BD relaționale şi, în fine, le-au dobândit, dezvoltatorii de aplicații au ajuns să se mulțumească cu MySQL-ul. Astăzi, apatia practicienilor față de bazele de date O-R este evidentă, iar lipsa documentației este cronică. SQL Server este cel mai rece la avansurile tipurilor, ierarhiei de tipuri, metodelor, moştenirii şi polimorfismului. PostgreSQL are implementată doar o cvasiierarhie de tabele care se regăseşte, pe un plan dezvoltat, şi în DB2. DB2-ul este mai generos în materie de opțiuni O-R, dar nu am reuşit să aflu prea multe surse informaționale dedicate subiectului. Cel mai bine plasat este Oracle, atât ca opțiuni de lucru, cât şi ca documentație ([Oracle 2008-1]).

Melton şi Simon se opresc asupra a două tipuri-utilizator, *distincte* şi *structurate* ([Melton & Simon 2002], pp. 45-47). La acestea adaugă tipurile referință (*reference*) şi masiv (*array*) care pot fi incluse în tipurile structurate. *Tipurile distincte* sunt particularizări ale unor tipuri-sistem „clasice" (primitive). *Tipurile structurate* sunt cele mai apropiate de percepția obişnuită asupra tipurilor definite de utilizator (UDT – User Defined Types), fiind ingredientul esențial pentru crearea tabelelor „tipate"[1] (*typed table*) pe care le vom discuta în paragraful 19.3. În tabelele „tipate", fiecare linie reprezintă o instanță a unui tip (clasă) structurat, iar atributele tabelei sunt, de fapt, atributele tipului pe care a fost definită. O instanță a unui *tip referință* este un pointer către un obiect dintr-o clasă. Pentru o bază de date, este inportant de ştiut că ținta unei valori referință este o linie dintr-o tabelă „tipată". Tipurile referință pot fi folosite pentru instituirea uneor legături părinte-copil dintre două

---

1. Îmi cer scuze şi pentru barbarismul lingvistic *tabele tipate*, dar n-am găsit ceva mai rezonabil.

tipuri, fiind similare legăturii părinte-copil dintre două tabele din modelul relaţional, fără însă a include restricţia referenţială propriu-zisă. Un *masiv* este o structură ce conţine o colecţie ordonată de elemente (date), în care fiecare element poate fi referit/accesat prin poziţia (ordinea) sa în cadrul colecţiei. Fiecare element/componentă este definit pe un alt tip de date, primitiv (simplu) sau stucturat. Un masiv are o dimensiune maximă, definită explicit sau implicit (de limitele sistemului).

19.1. Tipuri distincte şi structurate de date

Tipurile distincte au o importanţă redusă în economia aplicaţiilor cu baze de date, nefiind implementate decât în DB2 şi SQL Server, iar în SQL Server, chiar dacă sunt implementate, nu pot fi folosite la crearea tabelelor. Ideea de bază este de a crea subtipuri ale unor tipuri de bază, pentru a evita compararea lor sau combinarea lor accidentală în expresii. Astfel, definim două tipuri structurate, *t_cantitate* şi *t_preţ*, pe care le folosim la crearea tabelei TAB_TEST:

```
CREATE TYPE t_cantitate AS DECIMAL (10,3) FINAL;
CREATE TYPE t_pret AS DECIMAL(9,2) FINAL;

CREATE TABLE tab_test (
 produs VACHAR(30),
 cantitate t_cantitate,
 pret_unit t_pret
 );
```

Fiind de tipuri diferite, atributele Cantitate şi Preţ_Unit nu pot fi comparate, interogarea următoare soldându-se cu un mesaj de eroare:

```
SELECT *
FROM tab_test
WHERE cantitate > pret_unit
```

Din păcate, nici interogarea următoare nu ar duce-o mai bine:

```
SELECT t.*, cantitate * pret_unit AS val
FROM tab_test t
```

produsul valorilor celor două atribute fiind posibil doar cu ajutorul funcţiei CAST:

```
SELECT t.*, CAST (cantitate AS DECIMAL(10,3)) *
          CAST (pret_unit AS DECIMAL(9,2)) AS val
FROM tab_test t
```

19.1.1. Tipuri distincte şi structurate în PostgreSQL

Comanda de creare a unui nou tip de date pentru BD curentă este CREATE TYPE. Cel care lansează (cu succes) comanda devine proprietarul tipului, tip care poate fi partajat de către toţi utilizatorii bazei. În afara colecţiilor, pe care le discutăm în paragraful 19.2.1, documentaţia PostgreSQL face trimitere la trei categorii de tipuri utilizator:

- tipuri compozite – echivalentele tipurilor structurate;
- tipuri enumerate (ENUM);
- tipuri de bază, derivate și care modifică tipurile standard implementate în PostgreSQL.

Tipurile *enumerate* (ENUM) sunt utile atunci când valorile posibile ale atributelor/variabilelor definite pe această categorie fac parte dintr-o listă, cum ar fi sexul, starea civilă, zilele săptămânii etc.

```
CREATE TYPE t_sex AS ENUM ('F', 'M');
CREATE TYPE t_regiune AS ENUM ('Banat', 'Dobrogea', 'Moldova',
   'Muntenia', 'Oltenia', 'Transilvania');
```

În privința tipurilor structurate, să începem cu un exemplu simplu, dedicat județelor din țara noastră:

```
CREATE TYPE t_judet AS (
        jud CHAR(2),
        judet VARCHAR(30),
        regiune t_regiune
        );
```

Continuăm cu un tip de date care ne va ajuta într-o funcție ce furnizează un set de înregistrări necesar unui raport:

```
CREATE TYPE facturi_detaliate AS (
        nrfact INTEGER,
        datafact DATE,
        codcl SMALLINT,
        dencl VARCHAR(30),
        linie SMALLINT,
        codpr SMALLINT,
        denpr VARCHAR(30),
        cantitate NUMERIC(10),
        pretunit NUMERIC(9,2),
        valfaratva NUMERIC(14,2),
        tva NUMERIC(13,2),
        valcutva NUMERIC(15,2)
        );
```

Funcția care folosește acest tip în clauza RETURNS este *f_facturi_detaliate_clienți* și este prezentată în listingul 19.1. Nu există niciun element de dificultate care să merite discuții suplimentare.

Listingul 19.1. O funcție de folosește tipul nou definit

```
CREATE OR REPLACE FUNCTION f_facturi_detaliate_clienti (codcl_ clienti.codcl%TYPE)
        RETURNS SETOF facturi_detaliate AS $$
    SELECT f.nrfact, datafact, f.codcl, dencl, linie, codpr,
        f_denpr(codpr), cantitate, pretunit, cantitate * pretunit,
        cantitate * pretunit * f_proctva(codpr),
        cantitate * pretunit * (1 + f_proctva(codpr))
```

```
    FROM clienti c
        INNER JOIN facturi f ON c.codcl = f.codcl
        INNER JOIN liniifact lf ON f.nrfact = lf.nrfact
    WHERE f.codcl = $1
$$ LANGUAGE SQL;
```

Folosirea funcţiei într-o interogare este, după cum ştiţi, banală:

SELECT * FROM f_facturi_detaliate_clienti (CAST (1002 AS SMALLINT)) t

Următorul exemplu are, oarecum, un caracter reparator. În capitolul 11, PostgreSQL a stat pe tuşă, nefiind dotat cu nicio opţiune/funcţie OLAP. De aceea, definim un tip şi o funcţie (listingul 19.2) care să se apropie de logica operatorului CUBE.

```
CREATE TYPE t_cub_client_produs_zi AS (
    client VARCHAR(35),
    produs VARCHAR(35),
    zi VARCHAR(35),
    vinzari NUMERIC (14,2),
    nr INTEGER
    );
```

Listingul 19.2. O funcţie cât un CUBE

```
CREATE OR REPLACE FUNCTION f_cub_clienti_produs_zi (
    datai_ DATE, dataf_ DATE) RETURNS SETOF t_cub_client_produs_zi AS
$$
    SELECT ' ' || DenCl, ' ' || DenPr, ' '|| DataFact,
            SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
            COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl, DenPr, DataFact
        UNION
    SELECT ' ' || DenCl, ' ' || DenPr, 'subtotal client/produs',
            SUM(Cantitate * PretUnit * (1 + ProcTVA)), CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
            COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl, DenPr
        UNION
    SELECT ' ' || DenCl, 'subtotal client/zi' , ' ' || DataFact,
            SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01')
            AND COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl, DataFact
```

```
        UNION
    SELECT ' subtotal produs/zi', ' '||DenPr, ' ' || DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)), CAST (NULL AS INTEGER)
    FROM facturi
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
                COALESCE($1, DATE'2010-12-31')
    GROUP BY DenPr, DataFact
        UNION
    SELECT '  ' || DenCl, 'total client' , '***', SUM(Cantitate * PretUnit * (1 + ProcTVA)),
                CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl = facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
                COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl
        UNION
    SELECT ' total produs ', ' '||DenPr , '***',
                SUM(Cantitate * PretUnit * (1 + ProcTVA)), CAST (NULL AS INTEGER)
    FROM facturi
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
                COALESCE($1, DATE'2010-12-31')
    GROUP BY DenPr
        UNION
    SELECT ' ***', ' total zi' , ' ' ||DataFact,
                SUM(Cantitate * PretUnit * (1 + ProcTVA)), CAST (NULL AS INTEGER)
    FROM facturi
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
                COALESCE($1, DATE'2010-12-31')
    GROUP BY DataFact
        UNION
    SELECT ' = = TOTAL GENERAL = = ', ' = = = = = = = = = ' , ' = = = = = = = = = = ',
                SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
    FROM facturi
        INNER JOIN liniifact ON facturi.NrFact = liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr = produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
                COALESCE($1, DATE'2010-12-31')
    ORDER BY 1,2,3;
$$ LANGUAGE sql;
```

SELECT * FROM f_cub_clienti_produs_zi (NULL, NULL)

Suntem obligați să facem două precizări vecine cu dezmințirea. Mai întâi, funcția de mai sus nu este câtuși de puțin generală, referindu-se, strict, la facturi/vânzări. În al doilea rând, în loc de a crea tipul pentru a-l folosi în clauza RETURNS a funcției, puteam scrie direct RETURNS SETOF RECORD.

Ei, și acum ajungem la ceva mai interesant, ceea ce în limbajul contabililor se numește „fișă-șah". Noi o să-i spunem *situație matriceală a vânzărilor fiecărui produs pentru fiecare client*, adică ceva de genul raportului din figura 19.1.

client character varying	vnz_prod_1 numeric	vnz_prod_2 numeric	vnz_prod_3 numeric	vnz_prod_4 numeric	vnz_prod_5 numeric	vnz_prod_6 numeric	vnz_prod_8 numeric
Client 1 SRL	2361555.0000	3352077.0000			34025670.0000		571200.0000
Client 2 SA		956475.0000	11900.0000				
Client 3 SRL		332668.0000	266560.0000	568653.0000	45169425.0000		
Client 4		697000.5000			27740328.0000		
Client 5 SRL		746595.5000	397162.5000				
Client 6 SA		244923.0000		167260.5000	17652936.0000		
Client 7 SRL		1064385.0000					
Client destul de nou							

Figura 19.1. Un raport interesant

Mai întâi, redactăm o funcţie care să primească doi parametri, un cod de client şi un cod de produs şi să returneze suma vânzărilor din acel produs către clientul respectiv (vezi listingul 19.3).

Listingul 19.3. Funcţie simplă ce calculează vânzările dintr-un produs către un client

```
CREATE OR REPLACE FUNCTION f_client_produs (codcl_ SMALLINT, codpr_ SMALLINT,
        datai DATE, dataf DATE) RETURNS NUMERIC AS
$$
SELECT SUM(cantitate * pretunit * (1 + f_proctva($2)))
    FROM facturi f INNER JOIN liniifact lf ON f.nrfact = lf.nrfact
    WHERE codcl = $1 AND codpr = $2 AND
        DataFact BETWEEN COALESCE($3, DATE'2000-01-01') AND
            COALESCE($4, DATE'2010-12-31');
$$ LANGUAGE sql;
```

Chestiunea cea mai dificilă ţine de faptul că, în timp, în baza de date pot apărea clienţi şi produse noi, deci raportul din figura 19.1 nu are o structură fixă. De aceea, realizăm o funcţie-procedură care să creeze dinamic un tip ce stă la baza raportului, tip pe care-l vom denumi *t_temp*. Acesta va avea un prim atribut fix, *client*, şi un număr variabil de atribute, în funcţie de câte produse există în tabela cu acelaşi nume la momentul execuţiei funcţiei. Pentru asigurarea generalităţii, ne înfruptăm din SQL-ul dinamic.

Listingul 19.4. Funcţie-procedură ce creează, dinamic tipul T_TEMP

```
CREATE OR REPLACE FUNCTION f_create_type_t_temp () RETURNS VOID AS
$$
DECLARE
    v_sir VARCHAR (1000);
    rec_prod RECORD;
BEGIN
    EXECUTE 'DROP TYPE IF EXISTS t_temp CASCADE ';
    v_sir := 'CREATE TYPE t_temp AS (client VARCHAR ';
    FOR rec_prod IN SELECT CodPr FROM produse ORDER BY CodPr LOOP
        v_sir := v_sir || ', vnz_prod_' || LTRIM(TO_CHAR(rec_prod.CodPr,'999999')) || ' NUMERIC ';
    END LOOP;
    v_sir := v_sir || ') ';
    EXECUTE v_sir;
END;
$$ LANGUAGE plpgsql;
```

Apelăm funcţia:

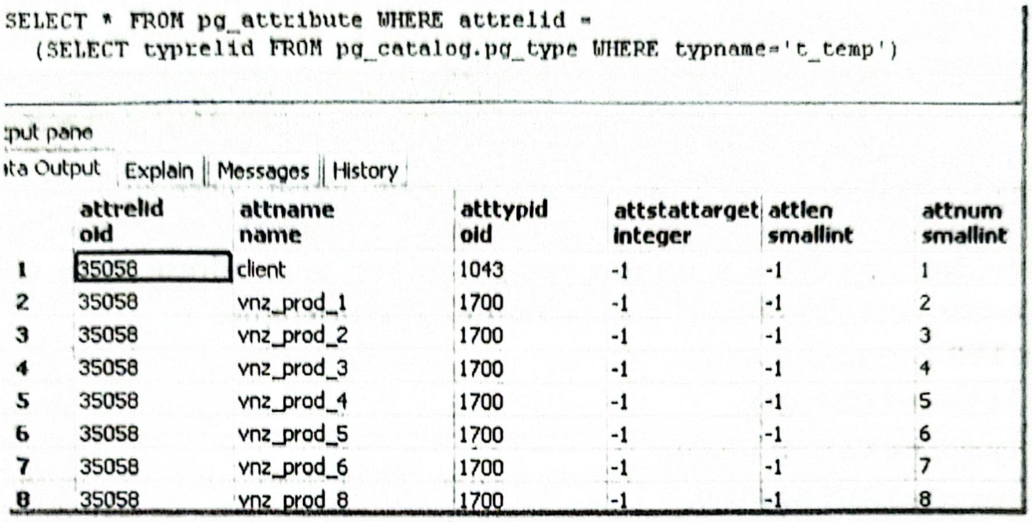
```
SELECT f_create_type_t_temp()
```

Clientul pgAdmin nu este prea darnic, aşa că, pentru a vedea structura tipului (figura 19.2) folosim dicţionarul de date:

SELECT * FROM pg_attribute

WHERE attrelid = (SELECT typrelid FROM pg_catalog.pg_type

WHERE typname = 't_temp')



```
SELECT * FROM pg_attribute WHERE attrelid =
  (SELECT typrelid FROM pg_catalog.pg_type WHERE typname='t_temp')
```

:put pane

ıta Output | Explain | Messages | History

	attrelid oid	attname name	atttypid oid	attstattarget integer	attlen smallint	attnum smallint
1	35058	client	1043	-1	-1	1
2	35058	vnz_prod_1	1700	-1	-1	2
3	35058	vnz_prod_2	1700	-1	-1	3
4	35058	vnz_prod_3	1700	-1	-1	4
5	35058	vnz_prod_4	1700	-1	-1	5
6	35058	vnz_prod_5	1700	-1	-1	6
7	35058	vnz_prod_6	1700	-1	-1	7
8	35058	vnz_prod_8	1700	-1	-1	8

Figura 19.2. Atributele tipului T_TEMP

Acum putem redacta o funcţie, pe care o vom denumi *f_dinamic_cl_prod* (vezi listingul 19.5). Mai întâi, construim şirul de caractere care, executat, va extrage conţinutul raportului. Apoi vom declara o variabilă cursor (*cursor1 REFCURSOR*) şi o deschidem specificând imensa frază SELECT stocată în şir (*OPEN cursor1 FOR EXECUTE v_sir*). Furnizarea, pe rând, a fiecărei înregistrări din variabila cursor se realizează prin *RETURN NEXT*.

Listingul 19.5. Funcţie ce returnează, la execuţie, raportul fin figura 19.1

```
CREATE OR REPLACE FUNCTION f_dinamic_cl_prod() RETURNS SETOF t_temp AS
$$
DECLARE
    v_sir VARCHAR (1000);
    rec_prod RECORD;
    rec_ RECORD;
    cursor1 REFCURSOR;
BEGIN
    v_sir := 'SELECT DenCl ';
    FOR rec_prod IN SELECT CodPr, DenPr FROM produse ORDER BY DenPr LOOP
        v_sir := v_sir || ', f_client_produs (codcl, CAST ('|| rec_prod.CodPr ||
            ' AS SMALLINT), null, null) AS "vnz_prod_'||rec_prod.DenPr ||'"';
    END LOOP;
    v_sir := v_sir || ' FROM clienti';

    OPEN cursor1 FOR EXECUTE v_sir;
    LOOP
        FETCH cursor1 INTO rec_;
        IF NOT FOUND THEN
            EXIT;
        END IF;
        RETURN NEXT rec_;
```

```
    END LOOP;

    RETURN;

END;
$$ LANGUAGE plpgsql;
```

Acum, la apelul funcţiei:

SELECT * FROM f_dinamic_cl_prod ()

avem toate şansele să obţinem raportul râvnit.

PostgreSQL nu permite definirea ierarhiilor de tipuri, ci numai ierarhii de tabele, subiect pe care-l vom ignora, probabil pe nedrept, şi în DB2. Dacă în paragrafele următoare vom defini un tip rădăcină - *tip_adresă_generală* şi câteva subtipuri ale acestuia - *tip_adresă_sat, tip_adresă_oraş, tip_adresă_bloc* - în PostgreSQL, pentru evidenţa analitică a tuturor componentelor unei adrese vom crea unul singur în care vom „îngrămădi" toate atributele/proprietăţile:

```
CREATE TYPE tip_adresa AS (
        codpostal CHAR(6),
        localitate VARCHAR(25),
        comuna VARCHAR(30),
        judet t_judet,
        strada VARCHAR(35),
        nr VARCHAR(10),
        bloc VARCHAR(20),
        scara VARCHAR(10),
        etaj VARCHAR(10),
        apartament NUMERIC(4)
        );
```

19.1.2. Tipuri simple în Oracle

Oracle candidează cu cele mai mari şanse la categoria „cel mai darnic server BD în materie de lucru cu obiecte", după cum vom vedea până la finalul acestui capitol. În continuare vom crea doar câteva tipuri simple, fără metode asociate. Începem cu un tip banal, asociat codului numeric personal:

```
CREATE OR REPLACE TYPE tip_cnp AS OBJECT (
        cnp NUMBER(13)
        ) FINAL
```

Pentru a mai câştiga în atractivitate, introducem în discuţie o ierarhie de tipuri. Cazul asupra căruia ne vom opri este cel al adreselor localităţilor din ţara noastră. Există diferenţe semnificative între adresele de la sat şi cele de la oraş. Toate, însă, au un cod poştal, o denumire de localitate şi o denumire de judeţ, aşa că vom defini un tip rădăcină denumit *tip_adresă_generală* cu trei proprietăţi (atribute):

```
CREATE OR REPLACE TYPE tip_adresa_generala AS OBJECT (
```

codpostal CHAR(6),

localitate VARCHAR2(25),

judet VARCHAR2(25)

) NOT FINAL NOT INSTANTIABLE

Tipul este *non-final*, adică urmează să apară, cât de curând, subtipuri ale sale, și *neinstanțiabil*, în sensul că nicio variabilă sau atribut de tabelă nu vor putea fi definite pe *tip_adresă_generală*. Figura 19.3 surprinde eroarea declanșată la execuția blocului anonim din listingul 19.6, în care variabila *a* este declarată de *tip_adresă_generală* și se încearcă inițializarea sa.

Listingul 19.6. Bloc anonim ce testează vigilența type-istă a Oracle

```
-- bloc anonim care foloseste acest tip - genereaza eroare, deoarece tipul este NEINSTANTIABIL
DECLARE
    a tip_adresa_generala := tip_adresa_generala ('700505', 'Iasi', 'Iasi');
BEGIN
    DBMD_OUTPUT.PUT_LINE(a.Localitate);
END;
/
```

Continuăm cu un subtip al *tip_adresă_generală* dedicat reprezentării adreselor din mediul rural în care, de obicei, nu apare strada și numărul, dar apare comuna. Acesta se va numi *tip_adresă_sat*:

CREATE OR REPLACE TYPE tip_adresa_sat UNDER tip_adresa_generala (

comuna VARCHAR2(30)

) FINAL

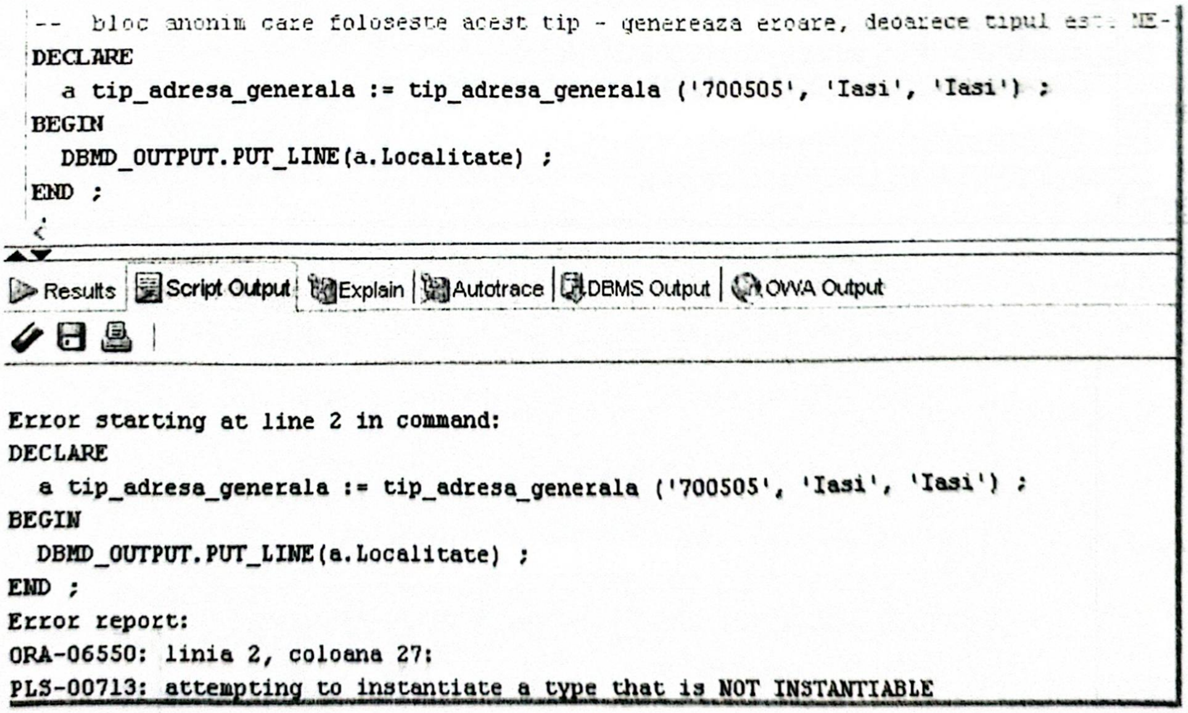


Figura 19.3. Încercare de a instanția un tip declarat NOT INSTANTIABLE

Celălalt subordonat direct al *tip_adresă_generală* este *tip_adresă_oraş* care, la rândul lui, are un tip-fiu – *tip_adresă_bloc*. Folosirea celor două tipuri ar fi justificată de faptul că în oraşe există persoane care locuiesc atât la bloc, cât şi la case (sau vile).

```
CREATE OR REPLACE TYPE tip_adresa_oras UNDER tip_adresa_generala (
        strada VARCHAR2(35),
        nr VARCHAR2(10)
        ) NOT FINAL
/
CREATE OR REPLACE TYPE tip_adresa_bloc UNDER tip_adresa_oras (
        bloc VARCHAR2(20),
        scara VARCHAR2(10),
        etaj VARCHAR2(10),
        apartament VARCHAR2(10)
 ) FINAL
/
```

Listingul 19.7 conţine un alt bloc PL/SQL anonim în care două variabile sunt declarate pe cele două (sub)tipuri, se instanţiază şi li se afişează (vezi figura 19.4) câte una dintre proprietăţi.

Listingul 19.7. Bloc anonim cu două variabile de *tip_adresă_oraş* şi *tip_adresă_bloc*

```
DECLARE
    a tip_adresa_oras := tip_adresa_oras ('700515', 'Pascani',
  'Iasi', 'Independentei', '12bis');
    b tip_adresa_bloc := tip_adresa_bloc ('700515', 'Pascani',
  'Iasi', 'Independentei', '12bis', 'H4', 'B', 4, 54);
BEGIN
    DBMS_OUTPUT.PUT_LINE(a.strada);
    DBMS_OUTPUT.PUT_LINE(b.apartament);
END;
```

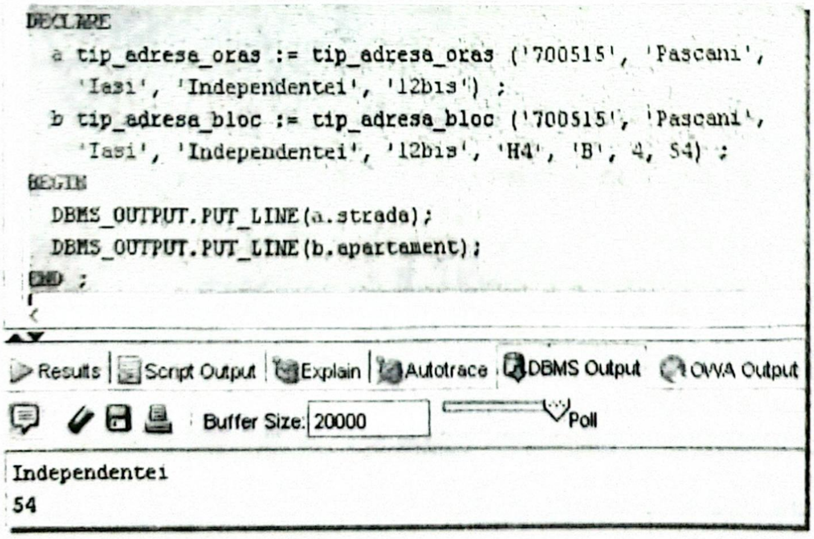


Figura 19.4. Două variabile de *tip_adresă_oraş* şi *tip_adresă_bloc*

Ierarhia declarată poate servi la definirea atributelor (proprietăţilor) unei tabele sau unui alt tip. Spre exemplu, *tip_persoana1* are patru proprietăţi, ultima, Adresa, fiind definită pe tipul-rădăcină al ierarhiei:

```
CREATE OR REPLACE TYPE tip_persoana1 AS OBJECT (
        cnp tip_cnp,
        nume VARCHAR2(30),
        prenume VARCHAR2(30),
        adresa tip_adresa_generala
        ) NOT FINAL
/
```

19.1.3. Tipuri simple în SQL Server

Versiunea folosită aproape exclusiv pe parcursul acestei lucrări, SQL 2005, este NULL-ă în materie de tipuri structurate. Tipurile pot fi create în limbaje ale platformei .NET şi folosite în SQL Server, dar ceea ce ne interesează pe noi sunt opţiunile SQL (T-SQL) prin care să declarăm tipuri mai simple sau mai complexe. SQL Server 2008 mai spală din ruşinea obiectual-relaţională a Microsoftului, însă noile opţiuni sunt de fandoseală, întrucât nici tipurile distincte, nici cele structurate, nu pot fi folosite la definirea atributelor în tabele, ceea ce este esenţial în BD O-R. Pentru tipurile distincte, sintaxa este destul de simplă:

```
CREATE TYPE t_sex FROM CHAR(1);
CREATE TYPE t_cantitate FROM NUMERIC(9,3);
CREATE TYPE t_pret FROM NUMERIC(10,2);
```

Şi tipurile structurate au sunt particulare SQL Server 2008, fiind declarate cu ajutorul clauzei TABLE:

```
CREATE TYPE t_coduri_cl AS TABLE (CodCl SMALLINT);

CREATE TYPE t_facturi AS TABLE (
        NrFact INTEGER,
        DataFact DATE,
        CodCl SMALLINT NOT NULL,
        Obs VARCHAR(50),
        ValTotala NUMERIC(10, 2) NULL,
        TVA NUMERIC(10, 2) NULL,
        ValIncasata NUMERIC(10, 2) NULL,
        Reduceri NUMERIC(9, 2) NULL,
        Penalizari NUMERIC(9, 2) NULL
        );
```

Ca şi în PostgreSQL, nu sunt permise ierarhiile de tipuri, deci nu putem vorbi despre moştenire sau alte ingrediente O-R. Vom mai discuta câte ceva despre aceste tipuri în paragraful 19.2.3.

19.1.4. Tipuri simple în DB2

Conform documentaţiei IBM ([IBM 2007-2]), în DB2 există patru tipuri de date definite de utilizator: distincte (*distinct*), structurate (*structured*), referinţă (*reference*) şi masiv (*array*). Pentru comparaţie cu exemplele din Oracle, în DB2 vom începe cu *tip_cnp*:

```
CREATE TYPE tip_cnp AS (
        cnp DECIMAL(13)
                ) MODE DB2SQL FINAL
```

Continuăm cu tipul-rădăcină al ierarhiei adreselor:

```
CREATE TYPE tip_adresa_generala AS (
        codpostal CHAR(6),
        localitate VARCHAR(25),
        judet VARCHAR(25)
        ) MODE DB2SQL NOT FINAL NOT INSTANTIABLE
```

şi subtipul său tip_adresă_sat:

```
CREATE TYPE tip_adresa_sat UNDER tip_adresa_generala AS (
        comuna VARCHAR(30)
        ) MODE DB2SQL FINAL;
```

De data aceasta, pentru testarea funcţionalităţii subtipurilor vom redacta câteva funcţii simple. Prima este *f_afişare_adresă_sat* şi face obiectul listingului 19.8.

Listingul 19.8. Funcţie SQL PL de afişare a unui şir de caractere ce conţine o adresă rurală

```
CREATE FUNCTION f_afisare_adresa_sat(adresa_ tip_adresa_sat)
   RETURNS VARCHAR(500)
  LANGUAGE SQL
  NO EXTERNAL ACTION
   RETURN 'CodPostal ' || adresa_..CodPostal || ', Comuna '|| adresa_..Comuna ||
       ', Sat ' || adresa_..Localitate || ', Judet ' || adresa_..Judet
```

La apelarea la, trebuie procedat cu grijă la specificarea instanţei-parametru:

```
SELECT f_afisare_adresa_sat ( tip_adresa_sat() ..codpostal('600515')
        ..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vanatori') )
FROM sysibm.dual
```

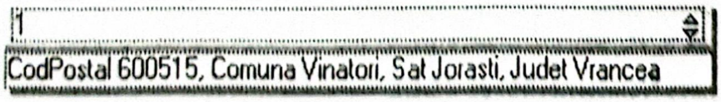


Figura 19.5. Apelarea funcţiei pentru afişarea unei adrese rurale

Continuăm cu *tip_adresă_oraş* şi funcţia destinată afişării unei adrese urbane de tip casă/vilă (listingul 19.9):

CREATE TYPE tip_adresa_oras UNDER tip_adresa_generala AS (
strada VARCHAR(35),
nr VARCHAR(10)
) MODE DB2SQL NOT FINAL

Listingul 19.9. Funcție SQL PL de afișare a adresei urbane de tip casă/vilă

```
CREATE FUNCTION f_afisare_adresa_oras(adresa_ tip_adresa_oras)
   RETURNS VARCHAR(500)
   LANGUAGE SQL
   NO EXTERNAL ACTION
   RETURN
      CASE WHEN adresa_..strada IS NOT NULL THEN 'Strada ' || adresa_..strada ELSE '' END ||
      CASE WHEN adresa_..nr IS NOT NULL THEN ' nr ' || adresa_..nr ELSE '' END ||
      ', CodPostal ' || adresa_..CodPostal || ', Localitate ' || adresa_..Localitate ||
      ', Judet ' || adresa_..Judet
```

Apelul funcției este similar celei precedente:

SELECT f_afisare_adresa_oras (tip_adresa_oras()
..codpostal('700515') ..localitate('Pascani')
..judet('Iasi') ..strada('Independentei') ..nr('12bis'))
FROM sysibm.dual

În fine, *tip_adresă_bloc* are și cea mai „voluminoasă" funcție de afișare (listingul 19.10):

CREATE TYPE tip_adresa_bloc UNDER tip_adresa_oras AS (
bloc VARCHAR(20),
scara VARCHAR(10),
etaj VARCHAR(10),
apartament DECIMAL(3)
) MODE DB2SQL FINAL

Listingul 19.10. Funcție SQL PL de afișare a adresei urbane de tip apartament

```
CREATE FUNCTION f_afisare_adresa_bloc(adresa_ tip_adresa_bloc)
   RETURNS VARCHAR(500)
  LANGUAGE SQL
  NO EXTERNAL ACTION
   RETURN
   CASE WHEN adresa_..strada IS NOT NULL THEN 'Strada ' || adresa_..strada ELSE '' END ||
   CASE WHEN adresa_..nr IS NOT NULL THEN ' nr ' || adresa_..nr ELSE '' END ||
   CASE WHEN adresa_..bloc IS NOT NULL THEN ' Bl. ' || adresa_..bloc ELSE '' END ||
   CASE WHEN adresa_..scara IS NOT NULL THEN ' Sc. ' || adresa_..scara ELSE '' END ||
   CASE WHEN adresa_..etaj IS NOT NULL THEN ' Etaj ' || adresa_..etaj ELSE '' END ||
   CASE WHEN adresa_..apartament IS NOT NULL THEN ' Ap. ' ||
      CAST (adresa_..apartament AS CHAR(3)) ELSE '' END ||
  ', CodPostal ' || adresa_..CodPostal || ', Localitate ' || adresa_..Localitate ||
  ', Judet ' || adresa_..Judet
```

SELECT f_afisare_adresa_bloc (tip_adresa_bloc() ..codpostal('700515')
..localitate('Pascani') ..judet('Iasi') ..strada('Independentei') ..nr('12bis')
..bloc('H4') ..scara('B') ..etaj ('4') ..apartament(54))
FROM sysibm.dual

Încheiem paragraful cu *tip_persoana1* ce foloseşte ierarhia de adrese pentru unul dintre atributele sale:

```
CREATE TYPE tip_persoana1 AS (
        cnp tip_cnp,
        nume VARCHAR(30),
        prenume VARCHAR(30),
        adresa tip_adresa_generala
        ) REF USING INTEGER MODE DB2SQL
```

19.2. Colecţii

Standardele SQL nu au fost grozav de generoase cu tipurile colecţie. Abia în SQL:1999 a apărut tipul ARRAY, iar în SQL:2003 este introdus tipul MULTISET. Dintre acestea doar primul este ceva mai prezent în SGBD-urile actuale. SQL Server este absent la capitolul colecţii, iar Oracle exagerat de generos.

19.2.1. Colecţii în PostgreSQL

În prezentarea colecţiilor PostgreSQL pornim de la o problemă plauzibilă. Să presupunem că avem nevoie de o listă a facturilor, filtrată pe clienţi. Până acum puteam să obţinem acest raport fie pentru toţi clienţii, fie numai pentru cei care îndeplineau condiţia formulată în clauza WHERE. În aplicaţii, utilizatorul ar putea dispune de o listă a clienţilor, din care ar bifa numai unul, doi, trei... clienţi care trebuie să apară în raportul respectiv. Cum folosirea operatorului IN poate fi anevoioasă (ce-ar fi dacă utilizatorul doreşte includerea în raport a 50 de clienţi din cei 1500 pe care îi are firma?), putem recurge la masive. Funcţia din listingul 19.11 este una simplă, însă atrage atenţia prin două noutăţi: parametrul de intrare este de tip ARRAY, iar clauza WHERE foloseşte opţiunea ANY.

Listingul 19.11. Funcţia SQL (PostgreSQL) ***f_facturi_filtrate*** cu un parametru de tip masiv

```
CREATE OR REPLACE FUNCTION f_facturi_clienti (coduri_clienti ANYARRAY)
        RETURNS SETOF facturi AS $$
    SELECT * FROM facturi WHERE codcl = ANY ( $1)
$$ LANGUAGE SQL;
```

Apelul funcţiei într-o interogare se face de maniera:

```
SELECT *
FROM f_facturi_clienti (ARRAY[1002, 1003, 1004] ) t
```

Putem, astfel, rescrie şi funcţia din listingul 19.1 - *f_facturi_detaliate_clienţi*, înlocuind parametrul simplu de intrare cu unul de tip ARRAY (listingul 19.12). Faptul că parametrul de intrare este de tip vector se indică prin clauza ANYARRAY, conferă un mare grad de generalitate (deşi nu era chiar necesar în exemplul nostru). Clauza WHERE a frazei SELECT ce constituie, de fapt, corpul funcţiei (SQL) prezintă predicatul *codcl = ANY ($1)*, echivalentul lui *codcl IN (coduri_clienţi(1), coduri_client(2)... coduri_clienţi(n))*.

Listingul 19.12. O funcţie ce foloseşte şi tipul nou definit şi un parametru-colecţie

```
CREATE OR REPLACE FUNCTION f_facturi_detaliate_clienţi (coduri_clienţi ANYARRAY)
        RETURNS SETOF facturi_detaliate AS $$
    SELECT f.nrfact, datafact, f.codcl, dencl, linie, codpr,
        f_denpr(codpr), cantitate, pretunit, cantitate * pretunit,
        cantitate * pretunit * f_proctva(codpr),
        cantitate * pretunit * (1 + f_proctva(codpr))
    FROM clienti c
        INNER JOIN facturi f ON c.codcl = f.codcl
        INNER JOIN liniifact lf ON f.nrfact = lf.nrfact
    WHERE f.codcl = ANY ($1)
$$ LANGUAGE SQL;
```

Folosirea sa într-o interogare este, de acum, banală:

SELECT * FROM f_facturi_detaliate_clienti (ARRAY[1001, 1002, 1004]) t

În PostgreSQL, tipurile ARRAY pot fi folosite la declararea proprietăţilor/atributelor altor tipuri:

```
CREATE TYPE tip_persoana AS (
        cnp CHAR(13),
        nume VARCHAR(30),
        prenume VARCHAR(30),
        adresa tip_adresa,
        telefoane CHAR(13) ARRAY[5],
        faxuri CHAR(13) ARRAY[5],
        emailuri VARCHAR(35) ARRAY[5]
        );
```

sau:

```
CREATE TYPE tip_telefoane AS (nrtel CHAR(10) ARRAY[10]);

CREATE TYPE tip_persoana2 AS (
        cnp CHAR(13),
        nume VARCHAR(30),
        prenume VARCHAR(30),
        adresa tip_adresa,
        telefoane tip_telefoane
        );
```

19.2.2. Colecţii în Oracle

În Oracle există trei categorii de colecţii: vectori asociativi (INDEX BY), tabele incluse (NESTED TABLE) şi vectori de mărime variabilă (VARRAY)[2]. Vectorii asociativi au fost utilizaţi în paragraful 17.3 pentru stocarea facturilor şi liniilor

---

2. Pentru detalii, vezi (şi) [Fotache *et al.* 2003].

şterse. Cel mai important dezavantaj al lor este că nu pot fi folosiţi la definirea atributelor în tabele. În schimb, tabelele incluse şi vectorii de mărime variabilă, da.

Mai întâi, să ne lichidăm de datoria din capitolul 16, legată de funcţii care returnează seturi de înregistrări. De fapt, cele două funcţii pe care le vom discuta îndată returnează colecţii de tip NESTED TABLES care pot fi convertite în seturi propriu-zise de înregistrări prin funcţia TABLE. Definim tipul *t_r_facturi* care are aceleaşi atribute ca tabela FACTURI:

```
CREATE TYPE t_r_facturi AS OBJECT (
        NrFact NUMBER(8),
        DataFact DATE,
        CodCl NUMBER(6),
        Obs VARCHAR2(50),
        ValTotala NUMBER(10,2),
        TVA NUMBER(10,2),
                ValIncasata NUMBER(10,2),
        Reduceri NUMBER(9,2),
        Penalizari NUMBER(9,2)
        );
```

Un tip colecţie NESTED TABLE se defineşte analog unui vector asociativ, eliminând clauza INDEX BY. Fiecare element al colecţiei *t_facturi* este un obiect de tipul *t_r_facturi*:

```
CREATE TYPE t_facturi AS TABLE OF t_r_facturi
```

Tipul colecţie este folosit de cele două funcţii din noul *pac_arhivare2* ce lucrează în tandem cu *pac_arhivare* din capitolul precedent. Ambele funcţii returnează tipul colecţie. Prima, *f_fuziune_facturi2*, returnează tot masivul odată, ceea ce poate congestiona traficul din reţea (dacă dimensiunea masivului este foarte mare). A doua este recomandată de documentaţia Oracle, întrucât componentele colecţiei se returnează una câte una (clauza PIPELINED).

Listingul 19.13. Un pachet ce „continuă" *pac_arhivare*

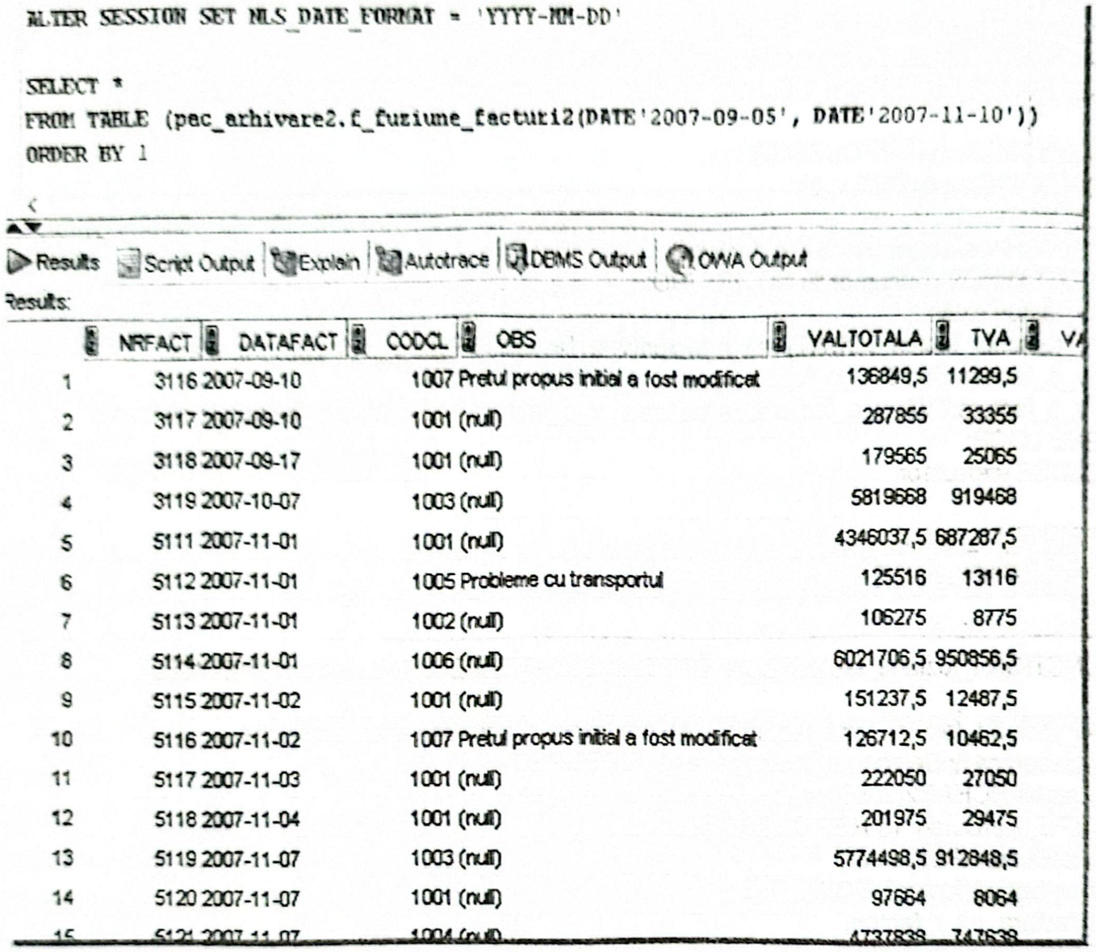
```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    -- pachetul pac_arhivare2 continuă operaţiunile demarate în paragraful 18.2 (SQL dinamic)
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
CREATE OR REPLACE PACKAGE pac_arhivare2 AUTHID CURRENT_USER AS
-- urmeaza alte doua functii pentru "recompunerea" dinamica a inregistrarilor arhivate

-- varianta 2 de reconstituire facturi : functie ce returneaza un NESTED TABLE;
-- in prealabil trebuie create tipurile T_R_facturi si T_facturi
FUNCTION f_fuziune_facturi2 (datai DATE, dataf DATE) RETURN t_facturi;

-- varianta 3 este o noua versiune a F_FUZIUNE_FACTURI2 cu variabile cursor si PIPELINED
FUNCTION f_fuziune_facturi3 (datai DATE, dataf DATE) RETURN t_facturi PIPELINED;

END pac_arhivare2;
-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
/
-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

```
  CREATE OR REPLACE PACKAGE BODY pac_arhivare2 AS
-- ==========================================================
---------------------------------------- -------------------------------------
FUNCTION f_fuziune_facturi2(datai DATE, dataf DATE) RETURN t_facturi
IS
 v_facturi t_facturi := t_facturi();
 v_sir VARCHAR2(2056) := ' ';
 TYPE trefcursor IS REF CURSOR;
 vrefcursor trefcursor;
 v_o_factura facturi%ROWTYPE;
BEGIN
 v_sir := pac_arhivare.f_sir_facturi(datai,   dataf);
 -- ar fi fost frumos, dar aceasta comanda nu functioneaza!!!
 --EXECUTE IMMEDIATE 'SELECT * BULK COLLECT INTO v_facturi FROM (' || v_sir || ')';

 -- asa ca folosim REFCURSOR
 OPEN vrefcursor FOR v_sir;
 LOOP
   FETCH vrefcursor INTO v_o_factura;
   EXIT WHEN vrefcursor % NOTFOUND;
   v_facturi.extend;
   v_facturi(v_facturi.COUNT) := t_r_facturi(v_o_factura.NrFact, v_o_factura.DataFact,
   v_o_factura.CodCl, v_o_factura.Obs, v_o_factura.ValTotala,
   v_o_factura.TVA, v_o_factura.ValIncasata, v_o_factura.Reduceri, v_o_factura.Penalizari);
 END LOOP;
 CLOSE vrefcursor;

 RETURN v_facturi;
END f_fuziune_facturi2;

------------------------------------------------------------------------------
FUNCTION f_fuziune_facturi3(datai DATE, dataf DATE) RETURN t_facturi PIPELINED
IS
 --v_facturi t_facturi; --:= t_facturi(); -- nu mai avem nevoie de variabila colectie,
 -- deoarece liniile se returneaza pe rand (PIPELINED)
 v_sir VARCHAR2(2056) := ' ';
 TYPE tRefCursor IS REF CURSOR;
 vRefCursor tRefCursor;
 v_o_factura facturi%ROWTYPE;
 v_factura_p t_r_facturi;
BEGIN
 v_sir := pac_arhivare.f_sir_facturi(datai, dataf);
 OPEN vrefcursor FOR v_sir;
 LOOP
   FETCH vRefCursor INTO v_o_factura;
   EXIT WHEN vRefCursor%NOTFOUND;
   v_factura_p := t_r_facturi(v_o_factura.NrFact, v_o_factura.DataFact,
     v_o_factura.CodCl,  v_o_factura.Obs,  v_o_factura.ValTotala,
     v_o_factura.TVA,  v_o_factura.ValIncasata, v_o_factura.Reduceri, v_o_factura.Penalizari);
   PIPE ROW(v_factura_p);
 END LOOP;
 CLOSE vrefcursor;
 RETURN;

END f_fuziune_facturi3;


END pac_arhivare2;
/
```

Dacă vă gândiţi la o modalitate de a „pasa" ceva mai simplu o linie din tabela facturi într-o instanţă a tipului *t_facturi*, aflaţi că m-am gândit şi eu, dar fără spor. Înainte de a demonstra funcţionalitatea celor două funcţii (vezi figura 19.6), trebuie să ne asigurăm că formatul datei calendaristice este cel ISO:

ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';

SELECT * FROM TABLE (pac_arhivare2.f_fuziune_facturi2(DATE'2007-09-05',
DATE'2007-11-10')) ORDER BY 1



	NRFACT	DATAFACT	CODCL	OBS	VALTOTALA	TVA
1	3116	2007-09-10	1007	Pretul propus initial a fost modificat	136849,5	11299,5
2	3117	2007-09-10	1001	(null)	287855	33355
3	3118	2007-09-17	1001	(null)	179565	25065
4	3119	2007-10-07	1003	(null)	5819668	919468
5	5111	2007-11-01	1001	(null)	4346037,5	687287,5
6	5112	2007-11-01	1005	Probleme cu transportul	125516	13116
7	5113	2007-11-01	1002	(null)	106275	8775
8	5114	2007-11-01	1006	(null)	6021706,5	950856,5
9	5115	2007-11-02	1001	(null)	151237,5	12487,5
10	5116	2007-11-02	1007	Pretul propus initial a fost modificat	126712,5	10462,5
11	5117	2007-11-03	1001	(null)	222050	27050
12	5118	2007-11-04	1001	(null)	201975	29475
13	5119	2007-11-07	1003	(null)	5774498,5	912848,5
14	5120	2007-11-07	1001	(null)	97664	8064
15	5121	2007-11-07	1004	(null)	4737838	747638

Figura 19.6. Testarea uneia dintre funcţiile ce returnează o colecţie

Acum putem redacta funcţia care să primească un parametru de tip *nested table* sau *varray* (parametru ce conţine oricâte coduri de client) şi care să returneze facturile corespunzătoare clienţilor-parametru. Rămâne să o realizaţi dumneavoastră în contul temei pentru acasă.

Pentru colecţii şi modul lor de folosire în tabele vom consuma câteva pagini în ultimul paragraf.

19.2.3. Colecţii şi SQL Server

Un titlu mai adecvat pentru acest paragraf ar fi fost *Nuca-n perete*, întrucât, aşa cum spuneam în deschiderea paragrafului 19.2, SQL Server este cel mai nevoiaş în materie de colecţii. Noroc cu variabilele de tip TABLE, cu ajutorul cărora putem

formula una dintre soluțiile la problema realizării unei funcții care să primească, drept parametru, mai multe coduri de client și care să returneze *recordset*-ul. Cum n-am văzut prin documentație să existe colecții, vom redacta, în deschidere, o funcție care primește un șir de caractere ce conține coduri de clienți separate prin virgulă și va returna o tabelă în care fiecare linie reprezintă un cod (vezi listingul 19.14). Figura 19.7 demonstrează că funcția este operațională.

Listingul 19.14. Funcție ce transformă un șir de caractere în înregistrările unei tabele

```
CREATE FUNCTION dbo.f_convert_sir_tab01 (
    @sir_coduri_cl_ VARCHAR(1000) )
    RETURNS @tabela TABLE (CodCl SMALLINT)
    AS
BEGIN
    DECLARE @sir_sursa VARCHAR(1000);
    DECLARE @i_inceput_cod SMALLINT;
    DECLARE @un_cod VARCHAR(10);
    DECLARE @i SMALLINT;
    SET @sir_sursa = LTRIM(RTRIM(@sir_coduri_cl_))
    SET @i = 2;
    SET @i_inceput_cod = 1;

    WHILE @i <=  LEN(@sir_sursa)
    BEGIN
        IF SUBSTRING(@sir_sursa, @i, 1) IN (' ', ',') OR @i = LEN(@sir_sursa)
            BEGIN
                    SET @un_cod = LTRIM(RTRIM(SUBSTRING(@sir_sursa, @i_inceput_cod,
                            @i - 1)))
                    IF @un_cod <> ''
                            INSERT INTO @tabela
                                    SELECT CAST (@un_cod AS SMALLINT)
                                    WHERE CAST (@un_cod AS SMALLINT) NOT IN
                                            (SELECT CodCl FROM @tabela)

                    SET @i_inceput_cod = @i + 1;
            END;
        SET @i = @i + 1;
    END;
    RETURN
END
```

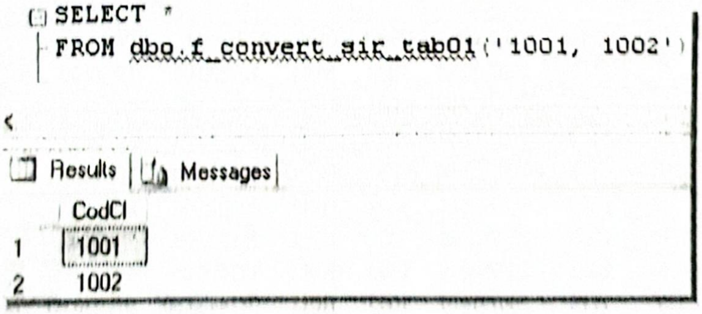


Figura 19.7. Testarea funcției ce convertește un șir de caractere în tabelă

Am avut probleme cu redactarea funcției care să primească tabela returnată de *f_convert_șir_tab01* drept parametru de intrare; de fapt, nu cu redactarea funcției, ci cu apelul său, întrucât, la apelare, argumentul său era funcția de mai sus. Până la urmă, sintaxa funcțională este cea din listingul 19.15.

Listingul 19.15. Funcție ce filtrează facturile după mai multe coduri de client

```
CREATE FUNCTION dbo.f_facturi_clienti (
    @sir_coduri_cl_ VARCHAR(1000) )
RETURNS @facturi TABLE
    (NrFact INTEGER,
    DataFact DATE,
    CodCl SMALLINT,
    Obs VARCHAR(50) ,
    ValTotala NUMERIC(10, 2),
    TVA NUMERIC(10, 2),
    ValIncasata NUMERIC(10, 2),
    Reduceri NUMERIC(9, 2),
    Penalizari NUMERIC(9, 2)
    )
    AS
BEGIN
--  DECLARE @coduri_cl t_coduri_cl -- nu functioneaza!!!!
    DECLARE @coduri_cl TABLE ( CodCl SMALLINT )

    INSERT INTO @coduri_cl
        SELECT * FROM dbo.f_convert_sir_tab01 (@sir_coduri_cl_)

    INSERT INTO @facturi
        SELECT * FROM facturi
        WHERE CodCl IN (SELECT DISTINCT CodCl FROM @coduri_cl)
    RETURN
END
```

Rezultatul execuției funcției pentru codurile de client 1002 și 1003 se prezintă în figura 19.8:

SELECT *

FROM dbo.f_facturi_clienti ('1002,1003')

ORDER BY 3, 1

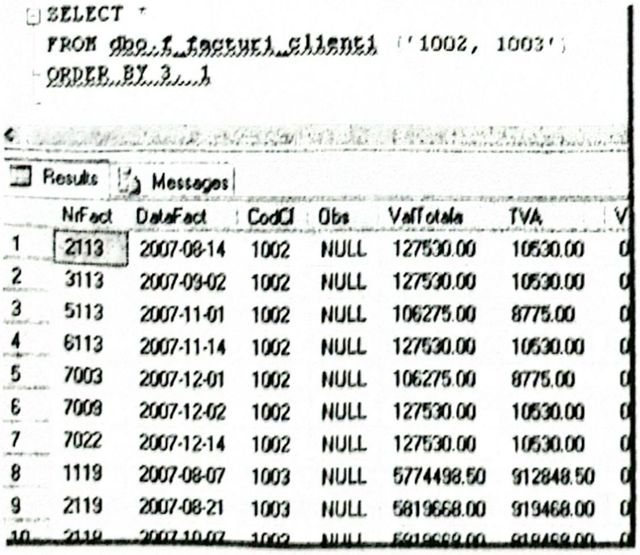


Figura 19.8. Testarea funcției ce filtrează facturile după mai multe coduri-client

Concluzia este că în SQL Server nu avem colecții, dar putem improviza câteva ceva.

19.2.4. Colecţii în DB2

SQL PL gestionează destul de elegant colecţiile de tip ARRAY, de fapt, singurul tip disponibil de colecţii. Declararea unui masiv care va conţine coduri de client se realizează astfel:

CREATE TYPE t_CodClArray as SMALLINT ARRAY[100]

În continuare redactăm procedura de filtrare a facturilor după mai multe coduri de client (şi două date calendaristice). Iniţial, am fi vrut să realizăm o funcţie, însă opţiunile de lucru cu masive sunt disponibile doar în proceduri. Ne reamintim procedura din listingul 16.64 pe care o redenumim şi o modificăm tocmai pentru a primi un parametru de tip ARRAY şi pentru a-l folosi în interogarea de definire a cursorului (vezi listingul 19.16). Funcţia ce transformă colecţia în seturi de înregistrări este UNNEST.

Listingul 19.16. Procedură SQL PL ce filtrează facturile după mai multe coduri de client

```
CREATE PROCEDURE p_fact_filtrate_array (coduriclienti_ t_CodClArray,
    data_initiala DATE, data_finala DATE)
    DYNAMIC RESULT SETS 1
P1: BEGIN
    -- Declararea cursorului
    DECLARE cursor1 CURSOR WITH RETURN FOR
        SELECT NrFact, DataFact, CodCl, Obs, ValTotala, TVA
        FROM  facturi
        WHERE CodCl IN (SELECT tCl.CodCl FROM UNNEST (coduriclienti_) AS tCl (CodCl) )
                    AND DataFact BETWEEN COALESCE(data_initiala, '2005-01-01')
                    AND COALESCE(data_finala, '2010-12-31');
    -- Cursorul ramane deschis pentru aplicatia client
    OPEN cursor1;
END P1
```

La apelul procedurii este necesar constructorul ARRAY:

CALL p_fact_filtrate_array (ARRAY[1002,1003],
CAST ('2007-10-07' AS DATE), CURRENT_DATE)

rezultatul fiind cel din figura 19.9.

Din păcate, în SQL PL, elementele unui masiv nu pot fi tipuri structurate.

CREATE TYPE tip_email AS (email VARCHAR(100)) MODE DB2SQL FINAL

funcţionează, nu însă şi

CREATE TYPE tip_telefon AS (numar CHAR(10)) MODE DB2SQL NOT FINAL;

CREATE TYPE tip_telefoane AS (tip_telefon) ARRAY[10];

Aşa că singura variantă este:

CREATE TYPE tip_telefoane AS CHAR(10) ARRAY[10];

```
CALL p_fact_filtrate_array ( ARRAY[1002,1003], CAST ('2007-10-07' AS DATE), CURRENT_DATE)

 --------------

 NRFACT      DATAFACT   CODCL  OBS                                                      VALTOTALA    TVA
 ----------- ---------- ------ -------------------------------------------------------- ------------ ------------
        3119 10/07/2007   1003 -                                                          5819668.00    919468.00
        4000 01/21/2009   1002 -                                                            21800.00      1800.00
        4001 01/22/2009   1002 -                                                            33700.00      3700.00
        4002 01/22/2009   1002 -                                                            33700.00      3700.00
        5113 11/01/2007   1002 -                                                           106275.00      8775.00
        5119 11/07/2007   1003 -                                                          5774498.50    912848.50
        6113 11/14/2007   1002 -                                                           127530.00     10530.00
        6119 11/21/2007   1003 -                                                          5819668.00    919468.00
        7003 12/01/2007   1002 -                                                           106275.00      8775.00
        7009 12/02/2007   1002 -                                                           127530.00     10530.00
        7013 12/07/2007   1003 -                                                          5774498.50    912848.50
        7017 12/07/2007   1003 -                                                          5819668.00    919468.00
        7022 12/14/2007   1002 -                                                           127530.00     10530.00
        7028 12/21/2007   1003 -                                                          5819668.00    919468.00

 14 record(s) selected.

 Return Status = 0
```

Figura 19.9. Testarea procedurii SQL PL ce filtrează facturile după mai multe coduri-client

O altă limită a DB2 este că tipurile colecţii nu pot fi folosite la declararea atributelor unui alt tip. Nici comanda următoare nu este acceptată:

```
CREATE TYPE tip_persoana AS (
        cnp tip_cnp,
        nume VARCHAR(30),
        prenume VARCHAR(30),
        adresa tip_adresa_generala,
        telefoane tip_telefoane,
        faxuri tip_telefoane,
        emailuri tip_emailuri
        ) MODE DB2SQL NOT FINAL
```

19.3. Stocarea tipurilor-utilizator în tabele

Toată discuţia legată de tipurile-utilizator, inclusiv de ierarhie, polimorfism, ar avea doar un caracter pitoresc dacă nu ar exista posibilitatea stocării lor în tabele. Putem vorbi despre două situaţii: 1) cea în care într-o tabelă, unul sau mai multe atribute sunt definite pe tipuri structurate şi 2) cea în care o linie (înregistrare) a unei tabele reprezintă o instanţă a unui tip structurat de date, aceasta din urmă fiind o tabelă *tipată*.

19.3.1. Tipuri şi tabele în PostgreSQL

În PostgreSQL, nu putem implementa ierarhii de tipuri, nu putem crea metode redactate PL/pgSQL şi nici tabele-tipate. În afara ierarhiilor de tabele (despre care

nu suflăm o vorbă), singura posibilitate este de a crea tabele în care atributele pot fi tipuri structurate sau colecţii:

```
CREATE TABLE persoane_ (
        cnp CHAR(13) PRIMARY KEY,
        nume VARCHAR(30),
        prenume VARCHAR(30),
        adresa tip_adresa,
        telefoane CHAR(13) ARRAY[5],
        faxuri CHAR(13) ARRAY[5],
        emailuri VARCHAR(35) ARRAY[5]
        );
```

Atributul Adresa din tabela PERSOANE_ a fost definit pe *tip_adresă*, iar atributele Telefoane, Faxuri şi Emailuri sunt colecţii. Scriptul din listingul 19.17 ilustrează modul de specificare a valorilor atributelor în comanda INSERT.

Listingul 19.17. Popularea cu înregistrări a unei tabele care are atribute definite pe tipuri structurate

```
DELETE FROM persoane_;

INSERT INTO persoane_ VALUES ('1760210390803', 'Popescu', 'Ioneliu',
    ROW ('123457', 'Jorasti', 'Vanatori', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    ARRAY ['0744123454', '0745123456', '0232222222'], -- telefoane
    NULL, -- n-are fax!
    ARRAY ['popescu_i@d.ro'] -- o adresa de email
    );
INSERT INTO persoane_ VALUES ('1660210390802', 'Ionescu', 'Vasile',
    ROW ('123456', 'Vanatori', 'Vanatori', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    ARRAY ['0744123455', '0788123456', '0232222225'],
    ARRAY ['0232222225'], -- fax
    ARRAY ['ionescu_v@d.ro', 'io201@yahoo.com']
    );
INSERT INTO persoane_ VALUES ('2821211390804', 'Iatu', 'Vasilica',
    ROW ('123458', 'Jaristea', 'Jaristea', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    NULL, -- n-are tel.
    NULL, -- n-are fax
    NULL -- n-are adrese de e-mail
    );
INSERT INTO persoane_ VALUES ('1781005390810', 'Cretu', 'Liviu',
    ROW ('700111', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Independentei',
        '11bis', 'G4', 'B', 'parter', 3),
    ARRAY ['0744293455', '0788123456', '0723128956', '0232277925'],
    ARRAY ['0232227725'],
    ARRAY ['cretu_liviu45@d.ro', 'cl6788@yahoo.com']
    );
INSERT INTO persoane_ VALUES ('2850419390812', 'Cretu', 'Livia',
    ROW ('700112', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Dependentei', '144', 'H14', 'D', '8', 45),
    ARRAY ['0744298855', '0788555456', '0728888956', '0235577925'],
    ARRAY['0232557725'],
    ARRAY ['cretu_livia@doi.ro', 'livia23@yahoo.com']
```

```
);
INSERT INTO persoane_ VALUES ('2831119390814', 'Bratu', 'Ioana',
    ROW ('700112', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Dependentei', '145', NULL, NULL,
        NULL, NULL),
    ARRAY ['0744298855', '0788777456', '0728999956', '0235577929'],
    NULL, -- fax
    NULL
);
```

La interogare, afişarea valorilor atributelor definite pe tipuri structurate şi colecţii este, în *pgAdmin*, cea din figura 19.10.

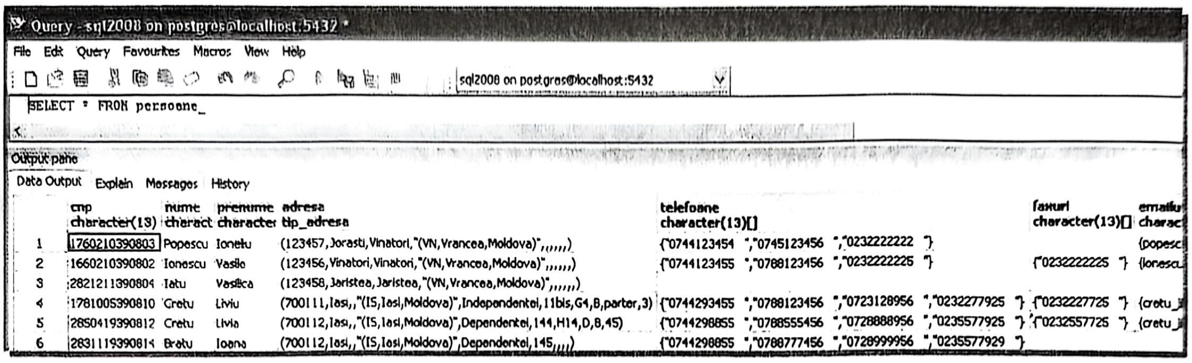


Figura 19.10. Afişarea conţinutului tabelei PERSOANE_

Extragerea poate fi limitată doar la componente ale colecţiilor. Astfel, ne interesează prima adresă de e-mail a fiecărei persoane (figura 19.11). Interogarea necesară este:

SELECT p.nume, p.prenume, p.emailuri[1] AS prima_adr_em

FROM persoane_ p

ORDER BY 1,2

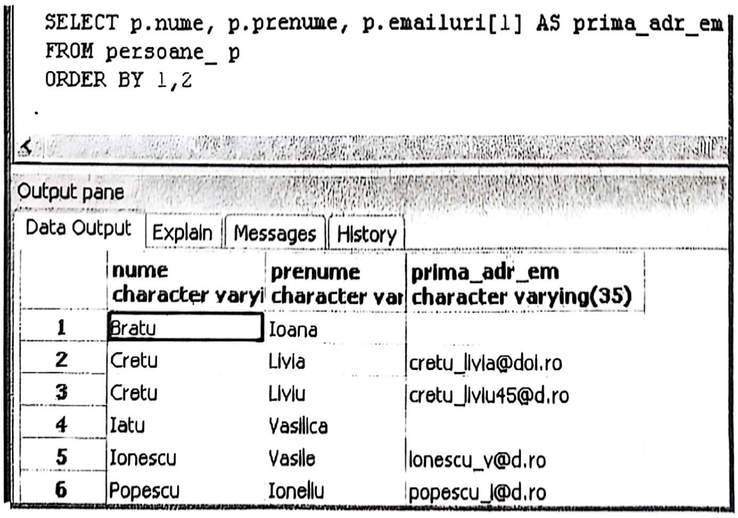


Figura 19.11. Prima adresă de e-mail a fiecărei persoane

Pentru căutarea în masiv, după cum am văzut în paragraful 19.2.1, se foloseşte operatorul ANY. Astfel, fraza SELECT ce răspunde la întrebarea *Cine are telefonul '0788555456'?* este:

SELECT p.nume, p.prenume, p.telefoane FROM persoane_ p

19.3.2. Tipuri şi tabele în Oracle

Cum avem deja un exemplu de tabelă în care câteva atribute sunt definite pe tipuri structurate şi pe tipuri colecţie, în Oracle ne grăbim să creăm o tabelă „tipată":

CREATE TABLE persoane1 OF tip_persoana1

Fiecare linie a acestei tabele va stoca o instanţă a clasei (tipului) *tip_persoana1* definit în paragraful 19.2.2. Listingul 19.18 conţine câteva comenzi INSERT prin care tabela PERSOANE1 este populată cu şase înregistrări.

Listingul 19.18. Popularea unei tabele „tipate"

```
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1660210390802),
        'Ionescu', 'Vasile',
        tip_adresa_sat(123456, 'Vanatori', 'Vrancea', 'Vanatori')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1760210390803),
        'Popescu', 'Ioneliu',
        tip_adresa_sat(123457, 'Jorasti', 'Vrancea', 'Vanatori')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2821211390804),
        'Iatu', 'Vasilica',
        tip_adresa_sat(123458, 'Jaristea', 'Vrancea', 'Jaristea')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1781005390810),
        'Cretu', 'Liviu',
        tip_adresa_bloc(700111, 'Iasi', 'Iasi', 'Independentei', '11bis',
            'G4', 'B', 'parter', 3)
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2850419390812),
        'Cretu', 'Livia',
        tip_adresa_bloc(700112, 'Iasi', 'Iasi', 'Dependentei', '144',
            'H14', 'D', '8', 45)
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2831119390814),
        'Bratu', 'Ioana',
        tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Dependentei', '144')
    ))
/
```

Actualizarea atributului Adresa definit pe tipul structurat *tip_adresă_generală* se realizează folosind constructorul subtipului care se referă la categoria adresei persoanei respective (în cazul nostru este vorba despre o adresă urbană pentru casă/vilă):

UPDATE persoane1 p

19.3.2. Tipuri și tabele în Oracle

Cum avem deja un exemplu de tabelă în care câteva atribute sunt definite pe tipuri structurate și pe tipuri colecție, în Oracle ne grăbim să creăm o tabelă „tipată":

CREATE TABLE persoane1 OF tip_persoana1

Fiecare linie a acestei tabele va stoca o instanță a clasei (tipului) *tip_persoana1* definit în paragraful 19.2.2. Listingul 19.18 conține câteva comenzi INSERT prin care tabela PERSOANE1 este populată cu șase înregistrări.

Listingul 19.18. Popularea unei tabele „tipate"

```
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1660210390802),
        'Ionescu', 'Vasile',
        tip_adresa_sat(123456, 'Vanatori', 'Vrancea', 'Vanatori')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1760210390803),
        'Popescu', 'Ioneliu',
        tip_adresa_sat(123457, 'Jorasti', 'Vrancea', 'Vanatori')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2821211390804),
        'Iatu', 'Vasilica',
        tip_adresa_sat(123458, 'Jaristea', 'Vrancea', 'Jaristea')
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (1781005390810),
        'Cretu', 'Liviu',
        tip_adresa_bloc(700111, 'Iasi', 'Iasi', 'Independentei', '11bis',
            'G4', 'B', 'parter', 3)
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2850419390812),
        'Cretu', 'Livia',
        tip_adresa_bloc(700112, 'Iasi', 'Iasi', 'Dependentei', '144',
            'H14', 'D', '8', 45)
    ))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
        tip_cnp (2831119390814),
        'Bratu', 'Ioana',
        tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Dependentei', '144')
    ))
/
```

Actualizarea atributului Adresa definit pe tipul structurat *tip_adresă_generală* se realizează folosind constructorul subtipului care se referă la categoria adresei persoanei respective (în cazul nostru este vorba despre o adresă urbană pentru casă/vilă):

UPDATE persoane1 p

SET adresa = tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Florilor', '7')

WHERE p.cnp.cnp = 2831119390814

SQL Developer-ul afişează conţinutului tabelei „tipate" de maniera celei din figura 19.14.

```
SELECT * FROM persoanel
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CNP	NUME	PRENUME	ADRESA
1	SQL2008.TIP_CNP(1660210390802)	Ionescu	Vasile	SQL2008.TIP_ADRESA_SAT(123456,Vinatori,Vrancea,Vinatori)
2	SQL2008.TIP_CNP(1760210390803)	Popescu	Ionelu	SQL2008.TIP_ADRESA_SAT(123457,Jorasti,Vrancea,Vinatori)
3	SQL2008.TIP_CNP(2621211390804)	Iatu	Vasilica	SQL2008.TIP_ADRESA_SAT(123458,Jaristea,Vrancea,Jaristea)
4	SQL2008.TIP_CNP(1761005390810)	Cretu	Liviu	SQL2008.TIP_ADRESA_BLOC(700111,Iasi,Iasi,Independentei,11bis,G4,B,parter,3)
5	SQL2008.TIP_CNP(2650419390812)	Cretu	Livia	SQL2008.TIP_ADRESA_BLOC(700112,Iasi,Iasi,Dependentei,144,H14,D,3,45)
6	SQL2008.TIP_CNP(2831119390814)	Bratu	Ioana	SQL2008.TIP_ADRESA_ORAS(700112,Iasi,Iasi,Florilor,7)

Figura 19.14. Afişarea conţinutului tabelei PERSOANE1

Pentru afişarea valorii unor atribute ale tipurilor structurate folosite la definirea *tip_persoane1*, acestea pot fi calificate astfel:

SELECT p.nume, p.prenume, p.cnp.cnp, p.adresa.Localitate, p.adresa.Judet

FROM persoane1 p

Rezultatul e cel din figura 19.15. A doua manieră va fi prezentată în paragraful 19.5 şi presupune invocarea metodelor definite la nivel de tip.

```
SELECT p.nume, p.prenume, p.cnp.cnp, p.adresa.Localitate,
  p.adresa.Judet
FROM persoanel p
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	NUME	PRENUME	CNP.CNP	ADRESA.LOCALITATE	ADRESA.JUDET
1	Ionescu	Vasile	1660210390802	Vinatori	Vrancea
2	Popescu	Ionelu	1760210390803	Jorasti	Vrancea
3	Iatu	Vasilica	2621211390804	Jaristea	Vrancea
4	Cretu	Liviu	1781005390810	Iasi	Iasi
5	Cretu	Livia	2650419390812	Iasi	Iasi
6	Bratu	Ioana	2831119390814	Iasi	Iasi

Figura 19.15. Calificarea atributelor din tipurile folosite la crearea tabelei PERSOANE1

Atributul Localitate face parte din tipul rădăcină al ierarhiei adreselor – *tip_adresă_generală*. La invocarea unui atribut de pe un subtip al rădăcinii, cum ar cazul comunei:

SELECT p.nume, p.prenume, p.adresa.judet AS judet,

p.adresa.comuna

FROM persoane1 p

Recepţionăm un binemeritat mesaj de eroare (vezi figura 19.16).

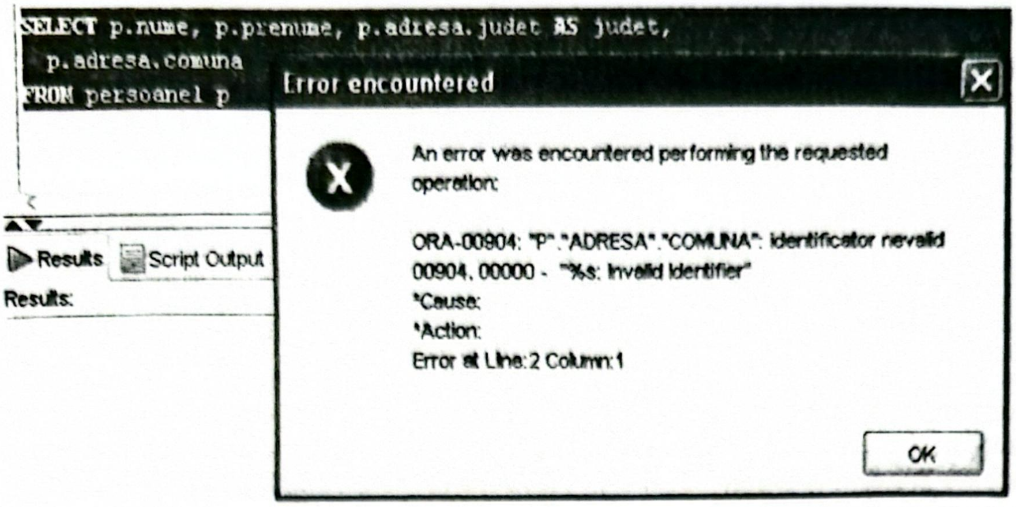


Figura 19.16. Eroare cauzată de accesarea unui atribut aflat pe o ramură a ierarhiei de adrese

Soluția ține de folosirea clauzei TREAT:

```
SELECT p.nume, p.prenume, p.adresa.judet AS judet,
        TREAT (p.adresa AS tip_adresa_sat).comuna
FROM persoane1 p
```

Filtrarea înregistrărilor din PERSOANE1, astfel încât să fie selectate numai persoanele care domiciliază la sate, este posibilă cu suportul operatorului IS OF:

```
SELECT p.nume, p.prenume, p.adresa.getJudet() AS Judet,
        TREAT (p.adresa AS tip_adresa_sat).getComuna() AS Comuna
FROM persoane2 p
WHERE p.adresa IS OF (tip_adresa_sat)
```

Înlocuirea lui *tip_adresă_sat* cu *tip_adresă_oraş* conduce la extragerea persoanelor cu adrese urbane, atât a celor de la casă/vilă, cât şi a celor de la bloc:

```
SELECT p.nume, p.prenume, p.adresa.Localitate,
        TREAT (p.adresa AS tip_adresa_oras).Strada
FROM persoane1 p
WHERE p.adresa IS OF (tip_adresa_oras)
```

Limitarea la cei care domiciliază la casă/vilă presupune combinarea operatorului IS OF cu clauza ONLY:

```
SELECT p.nume, p.prenume, p.adresa.Localitate,
TREAT (p.adresa AS tip_adresa_oras).Strada
FROM persoane1 p
WHERE p.adresa IS OF (ONLY tip_adresa_oras)
```

19.3.3. Tipuri şi tabele în SQL Server

Acesta este unul dintre paragrafele mele favorite. Nu avem ce discuta, întrucât SQL Server nu acceptă definirea atributelor din tabele pe tipuri-utilizator. Singurul

motiv pentru care am inclus această frază ține de respectarea numerotării paragrafelor, relativ la cele patru servere BD.

19.3.4. Tipuri şi tabele în DB2

Pe baza *tip_personal1* creat în paragraful 19.1.4, creăm o tabelă „tipată" după modelul celei din Oracle:

```
CREATE TABLE persoane1 OF tip_persoana1
        (REF IS OID USER GENERATED);
```

Popularea tabelei are câteva particularități ce decurg din obligativitatea folosirii identificatorului de obiecte (OID) (vezi listingul 19.19).

Listingul 19.19. Popularea tabelei „tipate" în DB2

```
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(1),
     tip_cnp() ..cnp(1660210390802) , 'Ionescu', 'Vasile',
          tip_adresa_sat() ..codpostal('123456')
               ..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vanatori')
     );
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(2),
     tip_cnp() ..cnp(1760210390803) , 'Popescu', 'Ioneliu',
          tip_adresa_sat() ..codpostal('123457')
               ..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vanatori')
     );
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(3),
     tip_cnp()..cnp(2821211390804) , 'Iatu', 'Vasilica',
          tip_adresa_sat() ..codpostal('123458')
               ..localitate('Jaristea') ..judet('Vrancea') ..comuna('Jaristea')
     );
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(4),
     tip_cnp() ..cnp(1781005390810) , 'Cretu', 'Liviu',
          tip_adresa_bloc() ..codpostal('700111') ..localitate('Iasi') ..judet('Iasi') ..strada('Independentei')
               ..nr('11bis') ..bloc('G4') ..scara('b') ..etaj('parter') ..apartament(3)
     );
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(5),
     tip_cnp() ..cnp(2850419390812) , 'Cretu', 'Livia',
          tip_adresa_bloc() ..codpostal('700112') ..localitate('Iasi') ..judet('Iasi') ..strada('Dependentei')
               ..nr('144') ..bloc('H14') ..scara('D') ..etaj('8') ..apartament(45)
     );
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(6),
     tip_cnp() ..cnp(2831119390814) , 'Bratu', 'Ioana',
          tip_adresa_oras() ..codpostal('700112') ..localitate('Iasi') ..judet('Iasi') ..strada('Dependentei')
               ..nr('144')
     );
```

Surprinzător este faptul că *DB2 Control Center* nu execută cea mai simplă formă de interogare a tabelei „tipate" (vezi figura 19.17).

```
SELECT p.* FROM personel p

------------------------------ Commands Entered ------------------------------
SELECT p.* FROM personael p;
------------------------------------------------------------------------------

SELECT p.* FROM persoanel p
SQL20015N  A transform group "DB2_PROGRAM" is not defined for data type
"MARIN.TIP_ADRESA_GENERALA".  SQLSTATE=42741

SQL20015N  A transform group "DB2_PROGRAM" is not defined for data type "MARIN.TIP_ADRESA_GENERALA
```

Figura 19.17. Imposibilitatea DB2 CC de a afişa înregistrările tabelei PERSOANE1

Putem, totuşi, afişa orice atribut folosind modalitatea din funcţiile realizate în paragraful 19.1.4:

```
SELECT p.nume, p.prenume, p.cnp..cnp, p.adresa..localitate, p.adresa..judet
FROM persoanel p
```

Similar celor discutate în Oracle, specificarea unui atribut sau metodă de pe o anumită ramură a ierarhiei poate fi realizată prin opţiunile TREAT şi IS OF. Astfel, pentru a afişa (şi) comunele persoanele cu adrese rurale, sintaxa este:

```
SELECT p.nume, p.prenume, p.cnp..cnp, p.adresa..localitate, p.adresa..judet,
        TREAT (p.adresa AS tip_adresa_sat)..comuna
FROM persoanel p
WHERE p.adresa IS OF (tip_adresa_sat)
```

19.4. Metode asociate tipurilor structurate în Oracle

Am recurs la această formă deşănţată de favoritism nu numai din comoditate. În PostgreSQL şi SQL Server nu există CREATE METHOD, iar pentru DB2, sursele bibliografice mi-au fost greu accesibile[3]. Aşa că voi ilustra tema propusă doar în Oracle, refolosind tipuri create în paragraful 19.3.2 şi adăugând, după nevoi, noi opţiuni şi operatori.

Primul tip pe care-l înzestrăm cu metode este *tip_cnp* (vezi listingul 19.20). Similar pachetelor PL/SQL, un tip are o parte de specificaţii publică şi un corp (privat) care conţine implementarea metodelor declarate în specificaţii.

Listingul 19.20. *Tip_cnp*, într-o nouă prezentare (cu metode)

```
DROP TYPE tip_cnp FORCE
/
CREATE OR REPLACE TYPE tip_cnp AS OBJECT (
    cnp NUMBER(13),
    MEMBER FUNCTION getCNP RETURN NUMBER,
    MEMBER FUNCTION getSex RETURN CHAR,
    MEMBER FUNCTION getDataNasterii RETURN DATE,
```

3. Afirmaţia este uşor exagerată. Cea mai bună documentare a lucrului cu tipuri şi tabele „tipate" am găsit-o în [IBM 2007-2], dar pe site-ul IBM sunt câteva articole interesante pe această temă, cum ar fi cel de la pagina: *http://www.ibm.com/developerworks/data/library/techarticle/dm-0506melnyk/*.

```
    MEMBER FUNCTION getVarstaCurentaAni RETURN NUMBER,
    MEMBER FUNCTION getVarstaCurentaAniLuni RETURN NUMBER,
    MEMBER FUNCTION getVarstaCurentaAniLuni2 RETURN INTERVAL YEAR TO MONTH,
    MEMBER FUNCTION getVarstaCurentaSir RETURN VARCHAR2,
    MEMBER FUNCTION getVarstaRelativaAni (data_calcul_ DATE) RETURN NUMBER,
    MEMBER FUNCTION getVarstaAni RETURN NUMBER,
    MEMBER FUNCTION getVarstaAni (data_calcul_ DATE) RETURN NUMBER -- supraincarcare
    ) FINAL
/

CREATE OR REPLACE TYPE BODY tip_cnp AS
    ------------------------------------------
    MEMBER FUNCTION getCNP RETURN NUMBER IS
    BEGIN
        RETURN SELF.cnp;
    END getCNP;
    ------------------------------------------
    MEMBER FUNCTION getSex RETURN CHAR IS
    BEGIN
        -- se analizeaza primul caracter din CNP
        RETURN
            CASE SUBSTR(SELF.cnp,1,1)
                    WHEN '1' THEN 'M'
                    WHEN '2' THEN 'F'
                    WHEN '5' THEN 'M'
                    WHEN '6' THEN 'F'
     WHEN '7' THEN 'M'
                    WHEN '8' THEN 'F'
                    ELSE NULL
            END;
    END getSex;
    ------------------------------------------
    MEMBER FUNCTION getDataNasterii RETURN DATE IS
        v_data DATE;
    BEGIN
        -- se decupeaza si interpreteaza ca data caracterele 2-7 din CNP
        v_data := TO_DATE(SUBSTR(SELF.cnp,2,6),'YYMMDD');

        IF SUBSTR(SELF.cnp,1,1) IN (1,2) THEN
            v_data := ADD_MONTHS(v_data, - 100 * 12);
        END IF;
        RETURN v_data;
    END getDataNasterii;
    -----------------------------------------------------
    MEMBER FUNCTION getVarstaCurentaAni RETURN NUMBER IS
    BEGIN
        -- facem apel la metoda de mai sus
        RETURN EXTRACT (YEAR FROM
            (CURRENT_DATE - SELF.getDataNasterii()) YEAR TO MONTH
                    );
    END getVarstaCurentaAni;
    ------------------------------------------
    MEMBER FUNCTION getVarstaCurentaAniLuni RETURN NUMBER IS
    BEGIN
        -- folosim doua metode anterioare
        RETURN SELF.getVarstaCurentaAni() +
            TRUNC ( (MONTHS_BETWEEN(CURRENT_DATE, SELF.getDataNasterii)
                    - SELF.getVarstaCurentaAni() * 12) * 0.01, 2);
    END getVarstaCurentaAniLuni;
```

```
    ----------------------------------------
    MEMBER FUNCTION getVarstaCurentaAniLuni2 RETURN INTERVAL YEAR TO MONTH IS
    BEGIN
        RETURN    (CURRENT_DATE - SELF.getDataNasterii()) YEAR TO MONTH;
    END getVarstaCurentaAniLuni2;
    ----------------------------------------
    MEMBER FUNCTION getVarstaCurentaSir RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.getVarstaCurentaAni() || ' ani si ' ||
            (SELF.getVarstaCurentaAniLuni() - SELF.getVarstaCurentaAni()) * 100
            || ' luni';

    END getVarstaCurentaSir;
    ----------------------------------------
    MEMBER FUNCTION getVarstaRelativaAni (data_calcul_ DATE) RETURN NUMBER IS
    BEGIN
        RETURN TRUNC(MONTHS_BETWEEN(data_calcul_, SELF.getDataNasterii() )/12,0);
    END getVarstaRelativaAni;
    ----------------------------------------
    MEMBER FUNCTION getVarstaAni RETURN NUMBER IS
    BEGIN
        RETURN SELF.getVarstaCurentaAni();
    END getVarstaAni;
    ----------------------------------------
    MEMBER FUNCTION getVarstaAni (data_calcul_ DATE) RETURN NUMBER
    -- metoda difera de precedenta prin semnatura (are un parametru)
    IS
    BEGIN
        RETURN SELF.getVarstaRelativaAni(data_calcul_);
    END getVarstaAni;
END;
/
```

Metodele asociate tipului valorifică încărcătura semantică a Codului Numeric Persoanal, extrăgând informații precum sexul și data nașterii și, pe baza lor, calculând vârsta curentă, exprimată în ani sau ani și luni, și vârsta relativă (vârsta pe care a avut-o sau o va avea „posesorul" CNP-ului la o anumită dată specificată).

Blocul anonim din listingul 19.21 și figura 19.18 sunt destinate a ilustra o modalitate de folosirea a metodelor acestui tip de date.

Listingul 19.21. Apelarea metodelor *tip_cnp*

```
-- un prima instantiere intr-un bloc PL/SQL a unui obiect de tip TIP_CNP
DECLARE
    a tip_cnp;
    b tip_cnp := tip_cnp(2871010370709);
BEGIN
    -- apelamm constructorul implicit
    a := tip_cnp (1660210390802);

    -- apelam metode ale tipului TIP_CNP
    DBMS_OUTPUT.PUT_LINE('Sex: ' || a.getSex());
    DBMS_OUTPUT.PUT_LINE('DataNasterii: ' || a.getDataNasterii());
    DBMS_OUTPUT.PUT_LINE('Varsta - ANI - la data curenta: ' || a.getVarstaCurentaAni());
    DBMS_OUTPUT.PUT_LINE('Varsta - ANI.LUNI - la data curenta: ' || a.getVarstaCurentaAniLuni());
    DBMS_OUTPUT.PUT_LINE('Varsta - ANI-LUNI2 - la data curenta: ' || a.getVarstaCurentaAniLuni2());
    DBMS_OUTPUT.PUT_LINE('Varsta (sir) - la data curenta: ' || a.getVarstaCurentaSir());
    DBMS_OUTPUT.PUT_LINE('Varsta (ani impliniti) - la 1 Ian 2005: ' ||
```

```
        a.getVarstaRelativaAni(DATE'2005-01-01'));
    -- testam supraincarcarea functiei getVarstaAni
    DBMS_OUTPUT.PUT_LINE('Varsta (ani impliniti) curenta: ' || a.getVarstaAni());
    DBMS_OUTPUT.PUT_LINE('Varsta (ani impliniti) - la 1 ian 2005: ' ||
        a.getVarstaAni(DATE'2005-01-01'));
END;
```

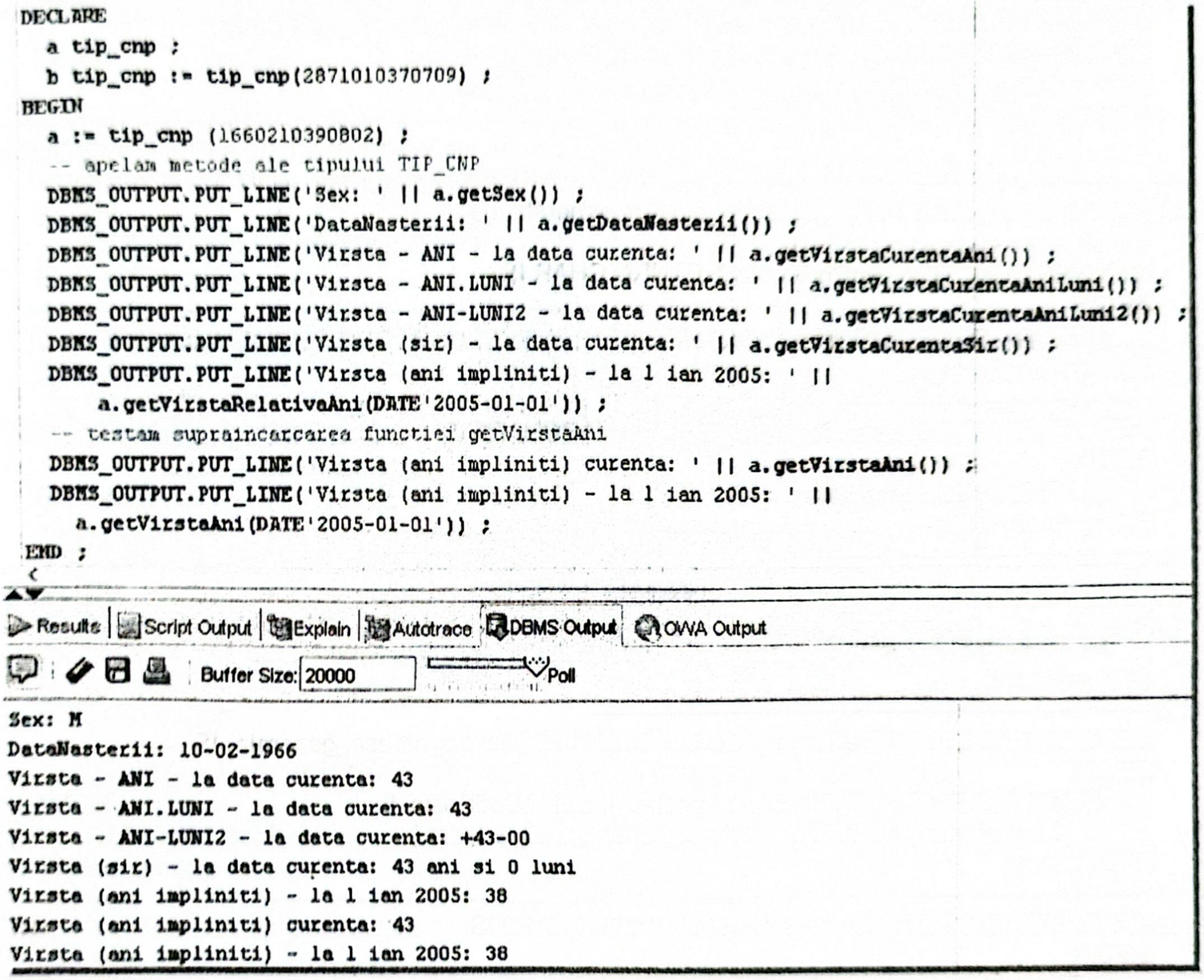


Figura 19.18. Apelul unui bloc anonim pentru verificarea metodelor *tip_cnp*

Continuăm cu declararea câtorva metode pentru tipul rădăcină destinat adreselor – *tip_adresă_generală* (vezi listingul 19.22). Fiind neinstanţiabil, declararea de metode-membru ar părea de prisos. Nu este chiar aşa, întrucât toate subtipurile sale vor moşteni atât atributele (proprietăţile), cât şi metodele. Dintre noutăţi, amintim:

- metodă de tip procedură (până aici şi de aici încolo, metodele sunt/vor fi preponderent funcţii), *afişare;*
- metodă de mapare, denumită *ordonare,* utilă la compararea a două instanţe a tipului (clasei) tip_adresa_generală (care, între noi fie vorba, n-o să existe niciodată, pentru că tipul este neinstanţiabil.

Listingul 19.22. Noile „dotări" ale *tip_adresă_generală*

```
-- 2. un tip de data compozit cu subtipuri -- ierarhie (probabil, cel mai celebru) -- ADRESA
-- ==================================================================
DROP TYPE tip_adresa_generala FORCE
/
```

```
CREATE OR REPLACE TYPE tip_adresa_generala AS OBJECT (
    codpostal CHAR(6),
    localitate VARCHAR2(25),
    judet VARCHAR2(25),
    MEMBER FUNCTION getCodPostal RETURN CHAR,
    MEMBER FUNCTION getLocalitate RETURN VARCHAR2,
    MEMBER FUNCTION getJudet RETURN VARCHAR2,
    MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_generala),
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
    ) NOT FINAL NOT INSTANTIABLE
/
-----------------------------------------------------------
CREATE OR REPLACE TYPE BODY tip_adresa_generala AS
    ---------------------------------
    MEMBER FUNCTION getCodPostal RETURN CHAR IS
    BEGIN
        RETURN SELF.CodPostal;
    END getCodPostal;
    ---------------------------------
    MEMBER FUNCTION getLocalitate RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Localitate;
    END getLocalitate;
    ---------------------------------
    MEMBER FUNCTION getJudet RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Judet;
    END getJudet;
    ---------------------------------------------------
    MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_generala) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Cod postal:' || SELF.CodPostal ||
            ', Localitate:' || SELF.localitate || ', judet:' || SELF.Judet );
    END afisare;
    ---------------------------------------------------
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN NULL;
    END afisare_sir;
    ---------------------------------------------------
    MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || codpostal;
    END ordonare;
    ---------------------------------------------------
END; -- TYPE BODY tip_adresa_generala
/
```

Continuăm cu *tip_adresa_sat*. În afara proprietăţii Comuna, noutăţile acestui (sub)tip sunt metoda funcţie ce retunează valoarea atributului Comuna (getComuna) şi suprascrierea procedurii *afişare* şi a funcţiilor *afişare_şir* şi *ordonare* (vezi listingul 19.22).

Listingul 19.22. „Reformatarea" *tip_adresă_sat*

```
-- = = = = = = = = = = = = = = = = subtipuri ale TIP_ADRESA_GENERALA = = = = = = = = = = = = = = = =
DROP TYPE tip_adresa_sat FORCE
/
```

```
-- pentru TIP_ADRESA_SAT introducem atributul COMUNA si suprascriem procedura de afisare
CREATE OR REPLACE TYPE tip_adresa_sat UNDER tip_adresa_generala (
    comuna VARCHAR2(30),
    MEMBER FUNCTION getComuna RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_sat),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
    )
    FINAL
/
------------------------------------------------------------------------
CREATE OR REPLACE TYPE BODY tip_adresa_sat AS
    MEMBER FUNCTION getComuna RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Comuna;
    END getComuna;
    ------------------------------------------------------------------------
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_sat) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Cod postal:' || SELF.CodPostal ||
        ', comuna:' || SELF.comuna || ', sat:' || SELF.localitate ||
            ', judet:' || SELF.Judet );
    END afisare;
    ------------------------------------------------------------------------
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN 'Cod postal:' || SELF.CodPostal ||
        ', comuna:' || SELF.comuna || ', sat:' || SELF.localitate ||
            ', judet:' || SELF.Judet;
    END afisare_sir;
    ------------------------------------------------------------------------
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || comuna || localitate;
    END ordonare;
    ------------------------------------------------------------------------
END; -- TYPE BODY tip_adresa_sat
/
```

Fiind un tip instanțiabil, putem testa modul în care funcționează metodele de tip procedură și funcție ale *tip_adresă_sat* (vezi listingul 19.23 și figura 19.19).

Listingul 19.23. Apelarea metodelor *tip_adresă_sat*

```
DECLARE
    a tip_adresa_sat := tip_adresa_sat ('600515', 'Jorasti', 'Vrancea', 'Vanatori');
    b tip_adresa_sat := tip_adresa_sat ('600514', 'Radulesti', 'Vrancea', 'Vanatori');
BEGIN
    DBMS_OUTPUT.PUT_LINE('Metoda 1 de afisare - procedura');
    -- apel metoda-procedura
    a.afisare;
    -- apel metoda-functie
    DBMS_OUTPUT.PUT_LINE('Metoda 2 de afisare - functie');
    DBMS_OUTPUT.PUT_LINE(a.afisare_sir);

    -- folosirea metodei de tip MAP pentru comparatie
    DBMS_OUTPUT.PUT_LINE(' = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ');
    DBMS_OUTPUT.PUT_LINE('a = ' || a.afisare_sir);
    DBMS_OUTPUT.PUT_LINE('b = ' || b.afisare_sir);
```

```
    IF a > b THEN
        DBMS_OUTPUT.PUT_LINE('a > b');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('a <= b');
    END IF;
END;
/
```

```
Metoda 1 de afisare - procedura
Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
Metoda 2 de afisare - functie
Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
=============================================
a=Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
b=Cod postal:600514, comuna:Vinatori, sat:Radulesti, judet:Vrancea
a <= b
```

Figura 19.19. Apelul unui bloc anonim pentru verificarea metodelor *tip_adresă_sat*

Fără comentarii suplimentare, în listingurile 19.24 şi 19.25 sunt recreate *tip_adresă_oraş* şi *tip_adresă_bloc,* cu metodele de rigoare.

Listingul 19.24. Noul *tip_adresă_oraş*

```
DROP TYPE tip_adresa_oras FORCE
/
CREATE OR REPLACE TYPE tip_adresa_oras UNDER tip_adresa_generala (
    strada VARCHAR2(35),
    nr VARCHAR2(10),
    MEMBER FUNCTION getStrada RETURN VARCHAR2,
    MEMBER FUNCTION getNr RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_oras),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
    )
    NOT FINAL
/
--------------------------------------------------
CREATE OR REPLACE TYPE BODY tip_adresa_oras AS
    ----------------------------------------
    MEMBER FUNCTION getStrada RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Strada;
    END getStrada;
    ----------------------------------------
    MEMBER FUNCTION getNr RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Nr;
    END getNr;
    ------------------------------------------------------------------
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_oras) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Str:' || SELF.Strada || ', nr:' || SELF.nr ||
        ', oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
        ', cod postal:' || SELF.CodPostal );
    END afisare;
    ------------------------------------------------------------------
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2       IS
    BEGIN
```

```
        RETURN 'Str:' || SELF.Strada || '. nr:' || SELF.nr ||
        ', oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
        ', cod postal:' || SELF.CodPostal;
    END afisare_sir;
    -----------------------------------------------------------
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || strada || nr;
    END ordonare;
    -----------------------------------------------------------
END; -- TYPE BODY tip_adresa_oras
/
```

Listingul 19.25. *Tip_adresă_bloc* într-o nouă prezentare

```
DROP TYPE tip_adresa_bloc FORCE
/
CREATE OR REPLACE TYPE tip_adresa_bloc UNDER tip_adresa_oras (
    bloc VARCHAR2(20),
    scara VARCHAR2(10),
    etaj VARCHAR2(10),
    apartament VARCHAR2(10),
    MEMBER FUNCTION getBloc RETURN VARCHAR2,
    MEMBER FUNCTION getScara RETURN VARCHAR2,
    MEMBER FUNCTION getEtaj RETURN VARCHAR2,
    MEMBER FUNCTION getApartament RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_bloc),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
    )
    FINAL
/
------------------------------------------------------------------------------
CREATE OR REPLACE TYPE BODY tip_adresa_bloc AS
    -----------------------------------------------------------
    MEMBER FUNCTION getBloc RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Bloc;
    END getBloc;
    ------------------------------------
    MEMBER FUNCTION getScara RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Scara;
    END getScara;
    ------------------------------------
    MEMBER FUNCTION getEtaj RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Etaj;
    END getEtaj;
    ------------------------------------
    MEMBER FUNCTION getApartament RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Apartament;
    END getApartament;
    -----------------------------------------------------------
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_bloc) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Str:' || SELF.Strada || ', nr:' || SELF.nr);
        DBMS_OUTPUT.PUT_LINE('      Bloc:' || SELF.bloc || ', scara:' || SELF.scara ||
            '           etaj:' || SELF.etaj || ', apart:' || SELF.apartament );
```

```
        DBMS_OUTPUT.PUT_LINE('    Oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
        ', cod postal:' || SELF.CodPostal );
    END afisare;
    ------------------------------------------------------------
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2       IS
    BEGIN
        RETURN 'Str.' || SELF.Strada || ', nr.' || SELF.nr ||', bl.' || SELF.bloc ||
            ', sc.' || SELF.scara || ', etaj ' || SELF.etaj || ', ap.' || SELF.apartament ||
            ', loc.' || SELF.localitate || ', jud.' || SELF.Judet ||
            ', cod ' || SELF.CodPostal;
    END afisare_sir;
    ------------------------------------------------------------
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || strada || nr || bloc || scara || apartament;
    END ordonare;
    ------------------------------------------------------------
END; -- TYPE BODY tip_adresa_bloc
/
```

Nu puteam lăsa de izbelişte tipurile colecţii, aşa că, pornind de la un tip „simplu", *tip_telefon*, vor crea un tip colecţie, *tip_telefoane*. Specificaţiile şi corpul primului sunt incluse în listingul 19.26. Metoda *getReţea* este, astăzi, mai puţin relevantă, întrucât clienţii pot schimba reţeaua de telefonie fără a modifica numărul.

Listingul 19.26. Specificaţiile şi corpul *tip_telefon*

```
DROP TYPE tip_telefon FORCE
/
CREATE OR REPLACE TYPE tip_telefon AS OBJECT (
    numar CHAR(10),
    MEMBER FUNCTION getNumar RETURN CHAR,
    MEMBER FUNCTION getRetea RETURN VARCHAR2,
    MEMBER FUNCTION getPrefixJud RETURN CHAR
    ) FINAL
/
CREATE OR REPLACE TYPE BODY tip_telefon AS
    ------------------------------
    MEMBER FUNCTION getNumar RETURN CHAR IS
    BEGIN
        RETURN SELF.Numar;
    END getNumar;
    ------------------------------
    MEMBER FUNCTION getRetea RETURN VARCHAR2 IS
        v_prefix CHAR(4);
    BEGIN
        v_prefix := SUBSTR(SELF.Numar,1,4);
        CASE
            WHEN SUBSTR(v_prefix,1,3) = '023' THEN -- ROMTELECOM
                    RETURN 'ROMTELECOM';
            WHEN SUBSTR(v_prefix,1,3) IN ('074', '075') THEN -- ORANGE
                    RETURN 'ORANGE';
            WHEN SUBSTR(v_prefix,1,3) IN ('072', '073') THEN -- VODAFONE
                    RETURN 'VODAFONE';
            WHEN SUBSTR(v_prefix,1,3) IN ('078') THEN -- Zapp
                    RETURN 'ZAPP';
        END CASE;
    END getRetea;
    ------------------------------------------------
```

```
    MEMBER FUNCTION getPrefixJud RETURN CHAR IS
    BEGIN
        IF SUBSTR(SELF.Numar,1,2) = '02' THEN
            RETURN SUBSTR(SELF.Numar,1,4);
        ELSE
            RETURN NULL;
        END IF;
    END getPrefixJud;
    ------------------------------------------------------
END; -- TYPE BODY tip_telefon
/
```

O persoană poate avea oricâte telefoane şi faxuri (numai să dispună de bani şi buzunare suficiente), aşa că apelăm la un tip de colecție din categoria NESTED TABLE:

DROP TYPE tip_telefoane FORCE;

CREATE OR REPLACE TYPE tip_telefoane IS TABLE OF tip_telefon;

Analog procedăm pentru adresele de poştă electronică:

DROP TYPE tip_email FORCE;

CREATE OR REPLACE TYPE tip_email AS OBJECT (email VARCHAR2(100));

DROP TYPE tip_emailuri FORCE;

CREATE OR REPLACE TYPE tip_emailuri IS TABLE OF tip_email;

Şi acum ajungem la tipul-concluzie al discuției noastre - *tip_persoană_complet* - definit ca un subtip al *tip_persoană1* din paragraful 19.1.2. Partea cea mai interesantă este prezența a două metode statice, una (*getNrPersoane*) care furnizează numărul de înregistrări ale tabelei PERSOANE3 şi a doua (*getCineAreTelefonul*), ce „cotrobăie" tot în PERSONAL3 după persoana ce are un număr de telefon parametru. Mai ciudat este că aceste două metode interoghează o tabelă care nu există şi pe care dorim să o creăm prin clauza *OF tip_persoana_complet*. Pentru acest gen de probleme, Oracle permite declararea incompletă a tipului. Mai întâi, declarăm numai specificațiie (vezi listingul 19.27).

Listingul 19.27. (Doar) Specificațiile *tip_persoană_complet*

```
DROP TYPE tip_persoana_complet FORCE
/
CREATE OR REPLACE TYPE tip_persoana_complet UNDER tip_persoana1 (
  telefoane tip_telefoane,
    faxuri tip_telefoane,
    emailuri tip_emailuri,
    MEMBER FUNCTION getNume RETURN VARCHAR2,
    MEMBER FUNCTION getPrenume RETURN VARCHAR2,
    MEMBER FUNCTION getNumePren RETURN VARCHAR2,
    MEMBER FUNCTION getToataAdresa RETURN tip_adresa_generala,
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    MEMBER FUNCTION getTelefoane RETURN tip_telefoane,
    MEMBER FUNCTION getFaxuri RETURN tip_telefoane,
    MEMBER FUNCTION getEmailuri RETURN tip_emailuri,
    STATIC FUNCTION getNrPersoane RETURN NUMBER,
    STATIC FUNCTION getCineAreTelefonul (tel_ tip_telefon) RETURN tip_persoana_complet
    ) NOT FINAL
/
```

Acum putem crea tabela „tipată":

CREATE TABLE persoane3 OF tip_persoana_complet
(PRIMARY KEY (cnp.cnp))
NESTED TABLE telefoane STORE AS telefoane_nt
NESTED TABLE faxuri STORE AS faxuri_nt
NESTED TABLE emailuri STORE AS emailuri_nt

după care se compilează corpul tipului (listingul 19.28).

Listingul 19.28. Corpul *tip_persoană_complet*

```
-- corpul tipului TIP_PERSOANA_COMPLET
CREATE OR REPLACE TYPE BODY tip_persoana_complet AS
    ------------------------------------------------
    MEMBER FUNCTION getNume RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Nume;
    END getNume;
    ------------------------------------------------
    MEMBER FUNCTION getPrenume RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Prenume;
    END getPrenume;
    ------------------------------------------------
    MEMBER FUNCTION getNumePren RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Nume || ' ' || SELF.Prenume;
    END getNumePren;
    ------------------------------------------------
    MEMBER FUNCTION getToataAdresa RETURN tip_adresa_generala IS
    BEGIN
        RETURN SELF.Adresa;
    END getToataAdresa;
    ------------------------------------------------
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.prenume || ' ' || SELF.nume || ', sex:' || SELF.cnp.getSex()
            || ', varsta:' || SELF.cnp.getVarstaAni () || ' ani, ' || SELF.adresa.afisare_sir;

    END afisare_sir;
  ------------------------------------------------
    MEMBER FUNCTION getTelefoane RETURN tip_telefoane IS
    BEGIN
        RETURN SELF.telefoane;
    END getTelefoane;
    ------------------------------------------------
    MEMBER FUNCTION getFaxuri RETURN tip_telefoane IS
    BEGIN
        RETURN SELF.faxuri;
    END getFaxuri;
    ------------------------------------------------
    MEMBER FUNCTION getEmailuri RETURN tip_emailuri IS
    BEGIN
        RETURN SELF.emailuri;
    END getEmailuri;
    ------------------------------------------------
    STATIC FUNCTION getNrPersoane RETURN NUMBER IS
        v_nr NUMBER(4) := 0;
```

```
    BEGIN
        SELECT COUNT(*) INTO v_nr FROM persoane3;
        RETURN v_nr;
    END getNrPersoane;
    ------------------------------------------------------------------------------
    STATIC FUNCTION getCineAreTelefonul (tel_ tip_telefon) RETURN tip_persoana_complet
    IS
        v_pers tip_persoana_complet;
    BEGIN
        SELECT object_value INTO v_pers FROM persoane3 p
        WHERE p.cnp.cnp IN
            (SELECT p.cnp.cnp
            FROM persoane3 p, TABLE (p.telefoane) t
            WHERE RTRIM(t.numar) = RTRIM(tel_.getNumar())  );
        RETURN v_pers;

    END getCineAreTelefonul;

END; -- TYPE BODY tip_persoana_complet
/
```

Inserarea înregistrărilor în PERSOANE3 se realizează în maniera prezentată în listingul 19.29.

Listingul 19.29. Inserarea de înregistrări în PERSOANE3

```
DELETE FROM persoane3;
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
        tip_cnp (1760210390803),
        'Popescu', 'Ioneliu',
        tip_adresa_sat(123457, 'Jorasti', 'Vrancea', 'Vanatori'),
        tip_telefoane (tip_telefon('0744123454'),
                       tip_telefon('0745123456'),
                       tip_telefon('0232222222')
                       ),
        NULL, -- n-are fax!
        tip_emailuri(tip_email('popescu_i@d.ro'))
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
        tip_cnp (1660210390802),
        'Ionescu', 'Vasile',
        tip_adresa_sat(123456, 'Vanatori', 'Vrancea', 'Vanatori'),
        tip_telefoane (tip_telefon('0744123455'), -- tel.
                       tip_telefon('0788123456'),
                       tip_telefon('0232222225')
                       ),
        tip_telefoane (tip_telefon('0232222225') -- fax
                       ),
        tip_emailuri(tip_email('ionescu_v@d.ro'), tip_email('io201@yahoo.com'))
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
        tip_cnp (2821211390804),
        'Iatu', 'Vasilica',
        tip_adresa_sat(123458, 'Jaristea', 'Vrancea', 'Jaristea'),
        NULL, -- n-are telefon!
        NULL, -- n-are fax!
        tip_emailuri() -- n-are adrese de e-mail!
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
```

```
        tip_cnp (1781005390810),
        'Cretu', 'Liviu',
        tip_adresa_bloc(700111, 'Iasi', 'Iasi', 'Independentei', '11bis',
            'G4', 'B', 'parter', 3),
        tip_telefoane (tip_telefon('0744293455'), -- telefon
                    tip_telefon('0788123456'),
                    tip_telefon('0723128956'),
                    tip_telefon('0232277925')
                    ),
        tip_telefoane (tip_telefon('0232227725') -- fax
                    ),
        tip_emailuri(tip_email('cretu_liviu45@d.ro'), tip_email('cl6788@yahoo.com'))
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
        tip_cnp (2850419390812),
        'Cretu', 'Livia',
        tip_adresa_bloc(700112, 'Iasi', 'Iasi', 'Dependentei', '144',
            'H14', 'D', '8', 45),
        tip_telefoane (tip_telefon('0744298855'), -- telefon
                    tip_telefon('0788555456'),
                    tip_telefon('0728888956'),
                    tip_telefon('0235577925')
                    ),
        tip_telefoane (tip_telefon('0232557725') -- fax
                    ),
        tip_emailuri(tip_email('cretu_livia@doi.ro'), tip_email('livia23@yahoo.com'))
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
        tip_cnp (2831119390814),
        'Bratu', 'Ioana',
        tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Dependentei', '144'),
        tip_telefoane (tip_telefon('0744298855'), -- telefon
                    tip_telefon('0788777456'),
                    tip_telefon('0728999956'),
                    tip_telefon('0235577929')
                    ),
        tip_telefoane (), -- fax
        tip_emailuri()
    ));
```

Dacă în paragraful 19.3.2 afişarea valorilor atributelor dintr-o tip structurat era realizată într-o manieră SQL-istă, prin „calificarea" numelor atributelor, acum putem redacta şi variante mai aproape de filosofia OO, în care valorile atributelor se obţin prin invocarea metodelor:

```
SELECT p.nume, p.prenume, p.cnp.getSex() AS Sex,
        p.cnp.getDataNasterii() AS DataNasterii,
        p.cnp.getVarstaCurentaSir() AS Varsta
FROM persoane3 p
```

În privinţă metodelor statice (cele care nu se aplică la nivel de instanţă, ci la nivel de clasă/tip), acestea pot fi invocate atât din blocuri anonime:

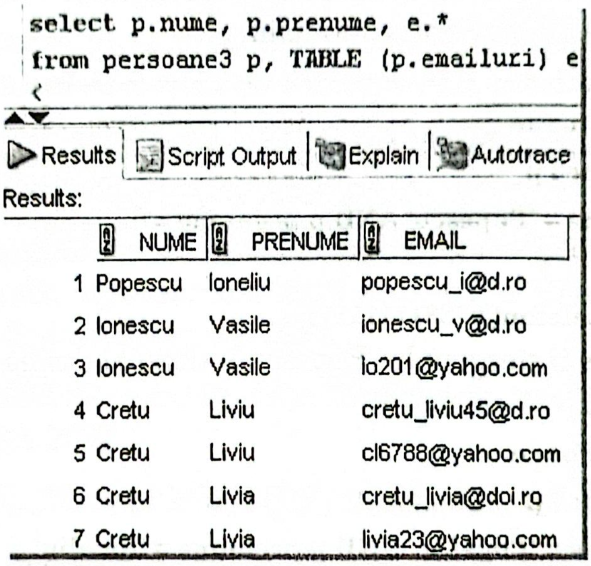
```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Nr inregistrarilor din tabela PERSOANE este '
        || tip_persoana_complet.getNrPersoane());
END;
```

cât şi dintr-o interogare:

SELECT tip_persoana_complet.getNrPersoane() FROM dual;

După cum am discutat în paragraful 19.2.2, conversia masivelor de tip NESTED TABLES şi VARRAYS în tabele presupune folosirea funcţiei TABLE. Astfel, pentru afişarea sub forma de tabelă a adreselor de e-mail ale tuturor persoanelor (figura 19.20), interogarea are forma:

```
SELECT p.nume, p.prenume, e.*
FROM persoane3 p, TABLE (p.emailuri) e
```



	NUME	PRENUME	EMAIL
1	Popescu	Ioneliu	popescu_i@d.ro
2	Ionescu	Vasile	ionescu_v@d.ro
3	Ionescu	Vasile	io201@yahoo.com
4	Cretu	Liviu	cretu_liviu45@d.ro
5	Cretu	Liviu	cl6788@yahoo.com
6	Cretu	Livia	cretu_livia@doi.ro
7	Cretu	Livia	livia23@yahoo.com

Figura 19.20. Afişarea sub formă de tabelă

Extragerea adreselor Liviei Creţu este posibilă în două variante:

```
SELECT e.*
FROM persoane3 p, TABLE (p.emailuri) e
WHERE nume = 'Cretu' AND prenume = 'Livia'
```

sau

```
SELECT *
FROM TABLE (
        SELECT p.emailuri
        FROM persoane3 p
        WHERE nume = 'Cretu' AND prenume = 'Livia'
        )
```

Pentru a afla persoanele care au numere de telefon în reţeaua ORANGE şi numerele respective, o sintaxă funcţională este:

```
SELECT p.nume, p.prenume, t.getNumar() AS Numar
FROM persoane3 p, TABLE (p.telefoane) t
WHERE t.getRetea() = 'ORANGE'
```

Pentru a pune la treabă cea de-a doua metodă statică, ne interesează numele de familie al persoanei care foloseşte telefonul cu numărul 0235577929, lucru posibil, după cum am văzut deja, atât printr-un bloc anomim:

```
DECLARE
    a tip_persoana_complet;
BEGIN
    a := tip_persoana_complet.getCineAreTelefonul(tip_telefon('0235577929'));
    dbms_output.put_line (a.nume);
END;
```

cât şi dintr-o interogare:

```
SELECT tip_persoana_complet.getCineAreTelefonul(tip_telefon('0235577929')).nume
FROM dual
```

Încheiem cu un exemplu de actualizare a colecţiilor. Adăugăm un număr de telefon din reţeaua ZAPP lui Popescu Ioneliu:

```
INSERT INTO TABLE (
        SELECT p.telefoane
        FROM persoane3 p
        WHERE p.nume = 'Popescu' AND p.prenume = 'Ioneliu'
                    )
        VALUES (tip_telefon('0788111111'));
```

şi, pentru compensare, îi ştergem lui Popescu Ioneliu al doilea număr Orange:

```
DELETE FROM TABLE (
        SELECT p.telefoane
        FROM persoane3 p
        WHERE p.nume = 'Popescu' AND p.prenume = 'Ioneliu'
        )
WHERE numar = '0745123456';
```

În fine, pentru a modifica prima adresă de e-mail a Liviei Creţu din 'cretu_livia@doi.ro' in 'cretu_livia@trei.com' sintaxa este:

```
UPDATE TABLE (SELECT p.emailuri FROM persoane3 p
              WHERE p.nume = 'Cretu' AND p.prenume = 'Livia'
              ) e
SET VALUE(e) = tip_email('cretu_livia@trei.com')
WHERE email = 'cretu_livia@doi.ro'
```

Au rămas multe alte ingrediente O-R nediscutate: tipuri REF, constructori expliciţi, tabele virtuale de obiecte, modificarea tipurilor etc. Din acest motiv, vă puteţi declara nemulţumiţi. Dacă, însă, luăm în considerare că niciuna dintre cărţile faimoase de SQL pe care le-am amintit în prefaţă nu oferă nici măcar o fracţiune din ceea ce am discutat în acest capitol, atunci avem motive să fim optimişti.

Bibliografie

[Astrahan *et al.* 1976] Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Wade, B.W., Watson, V., „System R: A Relational Approach to Database Management", *ACM Transactions on Database Systems*, vol. 1, nr. 2, iunie 1976

[Atzeni *et al.* 1999] Atzeni, P., Ceri, S., Paraboschi, P., Torlone, R., *Database Systems. Concepts, Languages and Architectures*, McGraw Hill, Londra, 1999

[Bain *et al.* 2003] Bain, T., Davidson, L., Dewson, R., Hawkins, C., *SQL Server 2000 Stored Procedures Handbook*, Apress, 2003

[Baklarz & Zikopoulos 2007] Baklarz, G., Zikopoulos, P. - *DB2 9 for Linux, Unix, and Windows : DBA guide, reference and exam prep*, ed. a VI-a, IBM Press/Pearson Education, Upper Saddle River, NJ, 2007.

[BDASEIG 2002] BDASEIG, *Baze de date. Fundamente teoretice şi practice*, Editura InfoMega, Bucureşti, 2002

[Beech 1989] Beech, D., „New Life for SQL", *Datamation*, 1, februarie 1989.

[Ben-Gan *et al.* 2006] Ben-Gan, I., Sarka, D., Wolter, R., *Inside Microsoft SQL Server 2005: T-SQL Programming*, Microsoft Press, 2006

[Birchall 2007] Birchall, G., *DB2 LUW v9.5. SQL Cookbook*. Disponibilă online la adresa: http://mysite.verizon.net/Graeme_Birchall/cookbook/DB2V95CK.PDF

[Bobrowski 2001] Bobrowski, S., „Creating Updatable Views", *Oracle Magazine*, martie-aprilie 2001

[Bedoya *et al.* 2004] Bedoya, H., Cruz, F., Lema, D., Singkorapoom, S., *Stored Procedures, Triggers and User Defined Functions – On DB2 Universal Database for iSeries*, IBM Redbooks, 2004

[Boyce *et al.* 1975] Boyce, R.F., Chamberlin, D.D., King, W.F., Hammer, M.M., „Specifying Queries as Relational Expresions: The SQUARE Data Sublanguage", *CACM*, 18, nr. 11, 1975

[Burleson *et al.* 2003] Burleson, D.K., Cook, J.P., Celko, J., *Advanced SQL Database Programmer Handbook*, Rampant Techpress, 2003

[Celko 1997] Celko, J., *Joe Celko's SQL Puzzles and Answers*, Morgan Kaufmann, San Francisco, 1997

[Celko 1999] Celko, J., *Joe Celko's Data and Databases: Concepts in Practice*, Morgan Kaufmann, San Francisco, 1999

[Celko 2000] Celko, J., *Joe Celko's SQL for Smarties: Advanced SQL Programming*, Morgan Kaufmann, San Francisco, 2000

[Celko 2004] Celko, J., *Joe Celko's Trees and Hierarchies in SQL for Smarties*, Elsevier/Morgan Kaufmann, San Francisco, 2004

[Celko 2005] Celko, J., *Celko's SQL for Smarties: Advanced SQL Programming*, ed. a III-a, Elsevier/Morgan Kaufmann, San Francisco, 2005

[Celko 2006] Celko, J., *Joe Celko's SQL Puzzles and Answers*, ed. a III-a, Morgan Kaufmann, San Francisco, 2007

[Chamberlin & Boyce 1974] Chamberlin, D., Boyce, R.F., „SEQUEL: A Structured English Query Language", ACM – SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, MI, mai 1974

[Chamberlin *et al.* 1976] Chamberlin, D. *et al.*, „SEQUEL 2: Definition, Manipulation and Control", *IBM J. Research and Development*, 20, nr. 6, noiembrie 1976

[Chamberlin 1980] Chamberlin, D., „A Summary of User Experience with the SQL Data Sublanguage", Proceedings of International Conference on Data Bases, Aberdeen, iulie 1980

[Chamberlin 1998] Chamberlin, D., „Super Groups", *DB2 Magazine On Line*, iarna 1998

[CODASYL 1969] CODASYL Systems Committee, *A Survey of Generalized Data Base Management Systems*, raport tehnic, mai 1969

[Codd 1969] Codd, E.F., „Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks", *IBM Research Report RJ*, 599, august 1969

[Codd 1970] Codd, E.F., „A Relational Model of Data for Large Shared Data Banks", *CACM*, vol. 13, nr. 6, 1970

[Codd 1979] Codd, E.F., „Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, vol. 4, nr. 4, decembrie 1979

[Codd 1981] Codd, E.F., „Data Models in Database Management", *ACM SIGMOD Record*, vol. 11, nr. 2, februarie 1981

[Codd 1989] Codd, E.F., „Fatal Flaws in SQL. Parts 1-2", *CMG Transactions*, vol. 1, nr. 1, 1989, pp. 29-33, 35-37

[Connolly & Begg 2002] Connolly, T., Begg, C.E., *Database Systems. A practical Approach to Design, Implementation and Management*, ed. a III-a, Addison-Wesley, Harlow, 2002

[Cumming & Russell 2007] Cumming, A., Russell, G., *SQL Hacks*, O'Reilly, Sebastopol, CA, 2007

[Darie *et al.* 2003] Darie, C., Watson, K., Hart, C., Hoffman, K., Skinner, J., *The Programmer's Guide to SQL*, Apress, 2003

[Date 1999-1] Date, C.J., „Thirty Years of Relational. Codd's Relational Algebra", *Intelligent Enterprise*, mai 1999

[Date 1999-2] Date, C.J., „Thirty Years of Relational: Relational Forever", *Intelligent Enterprise*, iunie 1999

[Date 2000] Date, C.J., *An Introduction to Database Systems*, ed. a VII-a, Addison-Wesley, Reading, MA, 2000

[Date 2004/2005] Date, C.J., *An Introduction to Database Systems*, ed. a VIII-a, Pearson Addison-Wesley, Boston, 2004. Ediţie tradusă în limba română cu titlul *Baze de date*, Editura Plus, Bucureşti, 2005

[Dollinger 1998] Dollinger, R., *Baze de date şi gestiunea tranzacţiilor*, ed. a II-a, Editura Albastră, Cluj-Napoca, 1998

[Dollinger & Andron 2004] Dollinger, R., Andron, L., *Utilizarea sistemului SQL Server*, Editura Albastră, Cluj-Napoca, 2004

[Douglas & Douglas 2005] Douglas, K., Douglas, S., *PostgreSQL: The comprehensive guide to building, programming, and administering PostgreSQL databases*, ed. a II-a, Sams Publishing, Indianapolis, IN, 2005

[Eisenberg *et al.*2004] Eisenberg, A., Kulkami, K., Melton, J., Michels, J.E., Zemke, F., „SQL:2003 Has Been Published", *ACM SIGMOD Record*, vol. 33, nr. 1, 2004

[Elmasri & Navathe 2000] Elmasri, R., Navathe, S.R., *Fundamentals of Database Systems*, Addison-Wesley, Reading, MA, 2000

[Faroult & Robson 2006] Faroult, S., Robson, P., *The Art of SQL*, O'Reilly, Sebastopol, CA, 2006

[Fehily 2006] Fehily, C., *SQL*, Editura All, Bucureşti, 2006

[Feuerstein 2007] Feuerstein, S., *Oracle PL/SQL Best Practices*, ed. a II-a, O'Reilly, 2007

[Fortier 1999] Fortier, P., *SQL-3. Implementing the Object-Relational Database*, McGraw-Hill, New York, 1999

[Fotache 1997] Fotache, M., *Baze de date relaţionale. Organizare, interogare şi normalizare*, Editura Junimea, Iaşi, 1997

[Fotache 2001] Fotache, M., *SQL. Dialecte DB2, Oracle şi Visual FoxPro*, Editura Polirom, Iaşi, 2001

[Fotache 2005] Fotache, M., *Proiectarea bazelor de date. Normalizare şi postnormalizare. Implementări SQL şi Oracle*, Editura Polirom, Iaşi, 2005

[Fotache 2007] Fotache, M., *Baze de date I*, Editura Universităţii „Al.I. Cuza", Iaşi, 2007

[Fotache *et al.* 2002] Fotache, M., Brava, I., Strîmbei, C., Creţu, L., *Visual FoxPro. Ghidul dezvoltării aplicaţiilor profesionale*, Editura Polirom, Iaşi, 2002

[Fotache *et al.* 2003] Fotache, M., Strîmbei, C., Creţu, L., *Oracle 9i2. Ghidul dezvoltării aplicaţiilor profesionale*, Editura Polirom, Iaşi, 2003

[Garcia-Molina *et al.* 2002] Garcia-Molina, H., Ullman, J., Widom, J., *Database Systems. The complete Book*, Prentice Hall, Upper Saddle Riner, NJ, 2002

[Geschwinde & Schönig 2001] Geschwinde, E., Schönig, H.-J., *PostgreSQL Developer's Handbook*, Sams Publishing, Indianapolis, IN, 2001

[Grama *et al.* 2006] Grama, A., Fotache, M., Ţugui, A., Dumitriu, F., *Instrumente software pentru afaceri*, Editura Universităţii „Al.I. Cuza", Iaşi, 2006

[Greenblatt & Waxman 1978] Greenblatt, D., Waxman, J., *A Study of Three Data Base Query Language: Improving Usability and Responsiveness*, Academic Press, 1978

[Gorman 1997] Gorman, M., „The Role of NIST in SQL Standardization", *The Data Administration Newsletter*, http://www.tdan.com/view-articles/4193, accesat la data de 10 mai 2005

[Greenwald *et al.* 2005] Greenwald, R., Stackowiak, R., Dodge, G., Klein, D., Shapiro, B., Chelliah, G.C., *Professional Oracle Programming*, Wiley, Indianapolis, IN, 2005

[Gorman 2001] Gorman, M., „Is SQL a Real Standard Anymore?", *The Data Administration Newsletter*, http://www.tdan.com/view-articles/4923/, accesat la data de 11 septembrie 2008

[Groff & Weinberg 2002] Groff, J., Weinberg, P., *SQL: The Complete Reference*, ed. a II-a, Osborne McGraw-Hill, New York, 2002

[Gulutzan & Pelzer 1999] Gulutzan, P., Pelzer, T., *SQL-99 Complete, Really*, R & D Books, Lawrence, KS, 1999

[Gulutzan 2005] Gulutzan, P., „Standard SQL", http://www.dbazine.com/db2/db2-disarticles/gulutzan3, accesat la data de 10 iunie 2006 (În prezent, articolul nu mai este disponibil.)

[Hainaut 1994] Hainaut, J.L., *Bases de données et modèles de calcul. Outils et méthodes pour l'utilisateur*, InterEditions, Paris, 1994

[Hernandez & Viescas 2000] Hernandez, M.J., Viescas, J.L., *SQL Queries for Mere Mortals*, Addison-Wesley, Boston, 2000

[Hurbean 2006] Hurbean, L., *Baze de date. Concepte teoretice şi abordare practică în Microsoft Access*, Editura Mirton, Timişoara, 2006

[IBM 2007-1] ***, *DB2 Version 9.5 for Linux, UNIX, and Windows. Developing User-defined Routines (SQL and External)*, International Business Machines Corporation, 2007

[IBM 2007-2] ***, *DB2 Version 9.5 for Linux, UNIX, and Windows. SQL Reference*, vol. 1, International Business Machines Corporation, 2007

[IBM 2007-3] ***, *DB2 Version 9.5 for Linux, UNIX, and Windows. SQL Reference*, vol. 2, International Business Machines Corporation, 2007

[Ionescu 2004] Ionescu, F., *Baze de date relaționale și aplicații*, Editura Tehnică, București, 2004

[ISO/IEC 2007-1] ISO/IEC JTC1/SC32, *WG3:BWU-002, H2-2008-006. Information Technology - Database Languages - SQL - Part 1: Framewowk (SQL Framework)*, WD 9075-1:200w(E), ed. Jim Melton, 2007

[ISO/IEC 2007-2] ISO/IEC JTC1/SC32, *WG3:BWU-003, H2-2008-007. Information Technology - Database Languages - SQL - Part 2: Foundation (SQL Foundation)*, WD 9075-2:200w(E), ed. Jim Melton, 2007

[ISO/IEC 2007-4] ISO/IEC JTC1/SC32, *WG3:BWU-005, H2-2008-009. Information Technology - Database Languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM)*, WD 9075-4:200w(E), ed. Jim Melton, Stephen Cannan, 2007

[Janmohamed *et al.* 2004] Janmohamed, Z., Liu, C., Bradstock, D., Chong, R., Gao, M., McArthur, F., Yip, P., *DB2 SQL PL, Second Edition: Deployment and Advanced Configuration Essential Guide for DB2 UDB on Linux., UNIX, Windows, i5/OS, z/OS*, Prentice Hall PTR, 2004

[Jones *et al.* 2005] Jones, A., Stephens, R.K., Plew, R.R., Garrett, R.F., Kriegel, A., *SQL Functions Programmer's Reference*, Wiley, Indianapolis, IN, 2005

[Kriegel & Trukhnov 2008] Kriegel, A., Trukhnov, B.M., *SQL Bible*, ed. a II-a, John Wiley & Sons, Indianapolis, IN, 2008

[Kulkarni 2003] Kulkarni, K., *Overview of SQL:2003*, http://www.wiscorp.com/SQL2003Features.pdf, accesat la 30 septembrie 2008

[Kyte 2005] Kyte, T., *Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions*, Apress, Berkeley, CA, 2005

[Lakshman 2003] Lakshman, B., *Oracle 9i PL/SQL: A Developer's Guide*, Apress, Berkeley, CA, 2003

[Larsen 1996] Larsen, S., „Powerful SQL: Beyond the Basics", *DB2 Magazine On Line*, iarnă 1996

[Larsen 1998] Larsen, S., „The SQL Double Double", *DB2 Magazine On Line*, primăvară 1998

[Lungu 2005] Lungu, I., *Baze de date Oracle. Limbajul SQL*, Editura Academiei de Studii Economice, Bucureşti, 2005

[Matthew & Stones 2005] Matthew, N., Stones, R., *Beginning Databases with PostgreSQL: From Novice to Professional*, ed. a II-a, Apress, Berkeley, CA, 2005

[McJones 1997] McJones, P. (ed.), *The 1995 SQL Reunion, People, Projects, and Politics*, SRC Technical Note 1997-018, 20 august 1997, http://www.research.digital.com/SRC/

[McLaughlin 2008] McLaughlin, M., *Oracle Database 11g PL/SQL Programming*, McGraw-Hill/Osborne, New York, 2008

[Melton 1996] Melton, J., „SQL3 and Objects: A new direction?", *Database Programming and Design*, august 1996

[Melton 2003] Melton, J., *Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann, San Francisco, CA, 2003

[Melton & Simon 2002] Melton, J., Simon, A.R., *SQL:1999. Understanding Relational Language Components*, Morgan Kaufmann, San Francisco, CA, 2002

[Molinaro 2006] Molinaro, A., *SQL Cookbook*, O'Reilly, Sebastopol, CA, 2006

[Mullins 2004] Mullins, C.S., *DB2 Developers's Guide*, Sams Publishing, Indianapolis, IN, 2004

[Oppel 2006] Oppel, A., *SQL fără mistere*, Editura Rosetti Educaţional, Bucureşti, 2006

[Oracle 2008-1] ***, *Oracle Database Object-Relational Developer's Guide 11g Release 1 (11.1)*, Oracle Corporation, 2008

[Oracle 2008-2] ***, *Oracle Database SQL Language, 11g Release 1 (11.1)*, Oracle Corporation, 2008

[Pascal 2000] Pascal, F., *Practical Issues in Database Management: A Reference for the Thinking Practitioner*, Addison-Wesley, 2000

[Pascu & Pascu 1994] Pascu C., Pascu A., *Totul despre... SQL. Interogarea bazelor de date*, Editura Tehnică, Bucureşti, 1994

[Price 2008] Price, J., *Oracle Database 11g SQL*, McGraw-Hill/Osbourne, New York, 2008

[Rahayu *et al.* 2006] Rahayu, W., Taniar, D., Pardede, E., *Object-Oriented Oracle*, CyberTech, Hershey, 2006

[Sitar-Tăut 2005] Sitar-Tăut, D.A., *Baze de date distribuite*, Editura Risoprint, Cluj-Napoca, 2005

[Sîrbu 2007] Sîrbu, F., *Rolul procedurilor stocate în aplicaţiile economice*, teză de doctorat, Universitatea „Al.I. Cuza" Iaşi, 2007

[Stonebraker *et al.* 1998] Stonebraker, M., Brown, P., Moore, D., *Object-Relational DBMSs*, ed. a II-a, Morgan Kaufmann, San Francisco, 1998

[Šunderić 2006] Šunderić, D., *Microsoft SQL Server 2005 Stored Procedure Programming in T-SQL & .NET*, ed. a III-a, McGraw-Hill/Osborne, San Francisco, 2006

[Troels 2008] Troels, A., *Comparison of different SQL implementations*, http://troels.arvin.dk/db/rdbms/, accesat la data de 30 septembrie 2008

[Tropashko 2005] Tropashko, V., „*Trees* in SQL: Nested Sets and Materialized Path", *DBAZine*, www.dbazine.com, 2005

[Türker & Gertz 2001] Türker, C., Gertz, M., „Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems", *VLDB Journal*, 10(4), 2001, pp. 241-269

[Urman *et al.* 2004] Urman, S., Hardman, R., McLaughlin, M., *Oracle Database 10g PL/SQL Programming*, McGraw-Hill/Osborne, New York, 2004

[Vieira 2007] Vieira, R., *Professional SQL Server 2005 Programming*, Wiley, Indianapolis, IN, 2007

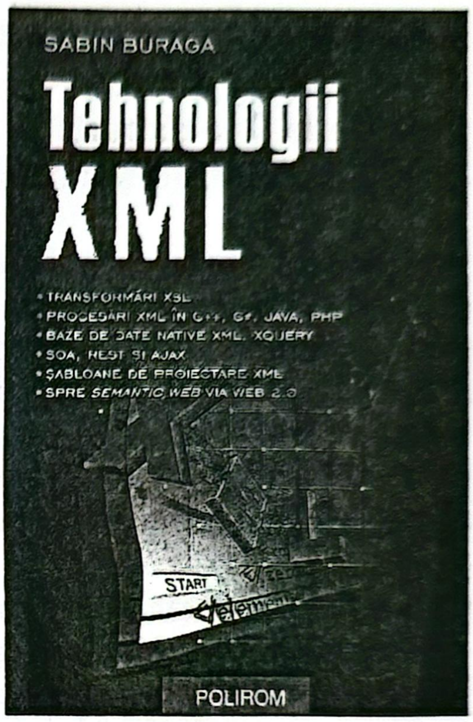
[Watt 2007] Watt, A., *Microsoft SQL Server 2005 Programming For Dummies*, Wiley, Hoboken, NJ, 2007

[Zemke *et al.* 1999-1] Zemke, F., Kulkarni, K., Witkowski, A., Lyle, B., „Introduction to OLAP functions", ISO/IEC JTC1/SC32 WG3:YGJ, ANSI NCITS H2-99-154, aprilie 1999

[Zemke *et al.* 1999-2] Zemke, F., Kulkarni, K., Witkowski, A., Lyle, B., „Proposal for OLAP functions", ISO/IEC JTC1/SC32 WG3:YGJ-nnn, ANSI NCITS H2-99-155r1, aprilie 1999

Sabin Buraga

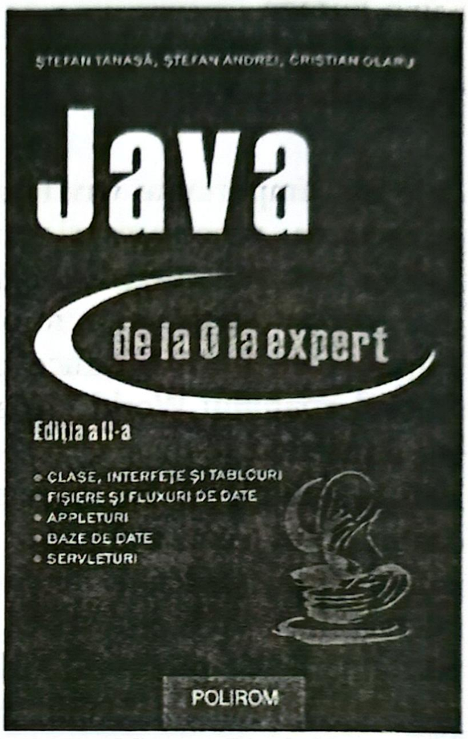
Tehnologii XML



Cele mai importante tendințe în utilizarea, proiectarea, procesarea şi conceptualizarea datelor prin suita de tehnologii XML (*Extensible Markup Language*) sunt obiectul acestui volum, care îl va conduce pe cititor spre viitorul spațiului World Wide Web. Subiectele atinse includ diverse aspecte ale familiei XML şi soluțiile pentru modelarea şi stocarea conținutului semistructurat, baze de date native XML, şabloane de proiectare XML, interacțiunea Web prin AJAX. Sunt descrise posibilitățile de procesare XML prin diferite metode, de redare a conținutului prin transformări XSL (*Extensible Stylesheet Language*), de validare a documentelor şi de creare a serviciilor Web, indicându-i-se cititorului cele mai bune practici de utilizare. De asemenea, cartea punctează conceptele de bază şi cele mai populare tehnologii ale Web-ului, subliniind direcțiile de evoluție către *Data Web*, *Social Web* şi ceea ce va fi *Semantic Web*.

Arhitectura spațiului WWW • Protocolul HTTP • *Cookie*-uri şi sesiuni • Date semistructurate • Familia XML: sintaxă, proiecte, transformări, procesare, validare • Servere de baze de date native XML • Interogări XQuery • Servicii Web • Transferul asincron prin AJAX • Aplicații Web fără fir • De la Web la *Data Web* şi *Semantic Web* • Specificarea meta-datelor şi ontologiilor Web • Inginerie Web

Ştefan Tanasă, Ştefan Andrei, Cristian Olaru



Java de la 0 la expert

Ediţia a II-a revăzută şi adăugită

Java este un limbaj de programare orientat obiect deja consacrat. Datorită dinamismului său este utilizat în scrierea celor mai multe aplicaţii distribuite, noile evoluţii tehnologice validând folosirea sa pentru software-ul dispozitivelor portabile (telefoane mobile, agende electronice, palmtop-uri).

Cititorului i se oferă un ghid complet de învăţare, volumul incluzând cunoştinţe de bază, dar şi de nivel mediu şi avansat referitoare la limbajul Java. Lucrarea este bogată în exemple de programe comentate, fiecare capitol terminându-se cu realizarea unei aplicaţii concrete, pentru a ilustra mai bine noţiunile şi instrumentele de programare puse la dispoziţie de acest limbaj. Capitolele au o structură similară, începând cu secţiunea de cuvinte-cheie şi încheind cu un scurt rezumat, un test-grilă şi exerciţii şi proiecte propuse spre implementare. Volumul, aflat la ediţia a doua revăzută şi adăugită, se adresează studenţilor şi profesorilor de la facultăţile de informatică şi tuturor celor pasionaţi de dezvoltarea de aplicaţii Web de anvergură.

Fundamentele limbajului Java • Fire de execuţie • Programarea reţelelor • Java Server Pages (JSP) • Procesarea documentelor XML • Interacţiunea cu utilizatorul prin Swing

www.polirom.ro

Redactor: Cătălina Popovici
Coperta: Laurenţiu Midvichi

Bun de tipar: martie 2009. Apărut: 2009
Editura Polirom, B-dul Carol I nr. 4 • P.O. Box 266, 700506, Iaşi
Tel. & Fax: (0232) 21.41.00; (0232) 21.41.11;
(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro
B-dul I.C. Brătianu nr. 6, et. 7, ap. 33; O.P. 37 •
P.O. Box 1-728, 030174, Bucureşti;
Tel.: (021) 313.89.78, E-mail: office.bucuresti@polirom.ro

Tiparul executat la S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, Bucureşti
Tel./Fax: 211.32.60, 212.29.27, E-mail: office@luminatipo.com

profesional practic

SQL

Dialecte DB2,
Oracle,
PostgreSQL
și SQL Server

Actualizată în conformitate cu versiunile recente ale standardului SQL şi îmbogăţită cu două dialecte foarte populare, lucrarea se adresează tuturor celor interesaţi de folosirea bazelor de date, constituind o importantă sursă de informaţii atât pentru profesioniştii din firmele/compartimentele IT, cât şi pentru studenţii la informatica, informatică economică, automatică şi calculatoare. Sunt prezentate implementările SQL a patru dintre cele mai populare servere de baze de date – IBM DB2, Oracle, PostgreSQL şi Microsoft SQL Server –, precum şi exemple practice pentru utilizatorii care doresc să obţină informaţii dintre cele mai diverse din bazele de date la care au acces. Volumul cuprinde numeroase probleme cu nivel diferit de dificultate, analogii şi soluţii detaliate, fiind atât un ghid de iniţiere în cele mai importante aspecte ale interogării bazelor de date, cât şi un suport pentru perfecţionare.

Despre bazele de date • Modelul relaţional de organizare a datelor • Standarde şi dialecte SQL. Tipuri de date. Crearea tabelelor şi modificarea conţinutului • Interogări SQL • Tabele temporare şi virtuale • Scheme, obiecte şi utilizatori • SQL dinamic • Obiectual-relaţional în SQL

Foto copertă: © by Chris Metcalf, © by Ejk

www.polirom.ro

ISBN 978-973-46-1339-7

9 789734 613397

POLIROM